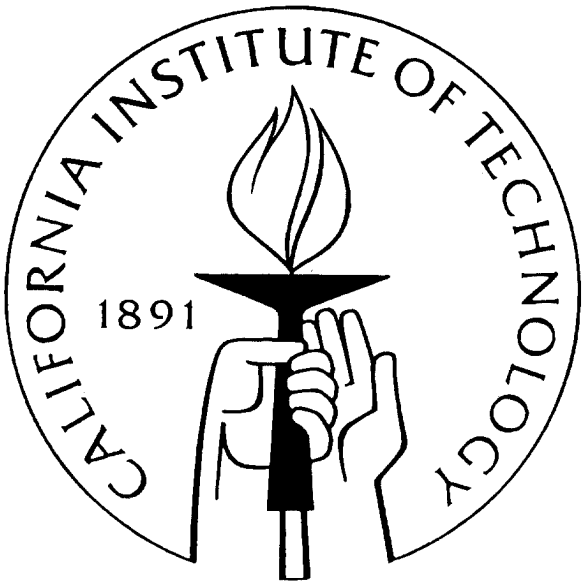

Combining Multilayer Networks to Combine Learning

Eric Bax

Computer Science Department
California Institute of Technology

Caltech-CS-TR-98-01

Combining Multilayer Networks to Combine Learning

Eric Bax

March 28, 1997

Contents

Abstract

This paper explores methods to combine networks, implementations of network combination on multicomputers, and applications of network combination. We divide the network combination process into two steps. First, a matching algorithm rearranges the networks so that corresponding weights in the networks being combined play corresponding roles in the networks' functions. Then a proportioning algorithm chooses a convex combination of the matched networks. The combination network is a weight-by-weight convex combination of the matched networks.

This paper begins by examining combinations of pairs of networks. Matching and proportioning algorithms are developed, analyzed, implemented, and tested. Next, algorithms are developed to combine several networks. Then the combination process is poked and prodded to explore its nature, its utility, and its limits.

Chapter 1

Introduction

Given a set of networks trained for the same task, there are two approaches to using the networks in concert to perform the task. Either each input can be fed to all networks, and their outputs can be combined in some fashion, or the networks themselves can be combined to form a single network which performs the task. By intuition, the first approach is more robust. Statistical tools can be applied to the ensemble of outputs, providing a degree of certainty regarding the final averaged output. However, there are tasks for which network combination is better suited than output fusion.

Fusion of network outputs can be seen as a process that produces a larger, more complex network from the initial set of networks. The early layers are composed of the set of networks, and the final layer performs the output fusion. If this growth process is iterated, large and complex networks are formed. In contrast, combining the networks themselves produces networks with the same architecture at every iteration. So combining networks allows repeated merging of learners. Thus, network combination is a useful tool for genetic algorithms and other training schemes in which learning is combined at intervals.

Iterated network combination can increase the speed of network training on message-passing multicomputers. First, consider the following scheme to learn with batch mode backpropogation using data parallelism [8, 9, 16, 19, 20]. The data is distributed among the processors, and there is a copy of the network in each processor. Concurrently, each processor calculates $\partial E/\partial w$ for its share of the data. Then processor $\partial E/\partial w$'s are summed to calculate $\partial E/\partial w$ for the entire data set, and the sum is broadcast to all processors. Finally, the weights are updated in each processor, completing a batch mode epoch (see Figure 1.1).

The bottleneck in this scheme is the global communication step required to sum $\partial E/\partial w$ across all processors. Communication can be reduced by training for more than one epoch (in sequential or batch mode) within each processor between communication phases, thus abandoning global batch mode. But as networks train within the processors, they drift apart in weight space, leaving

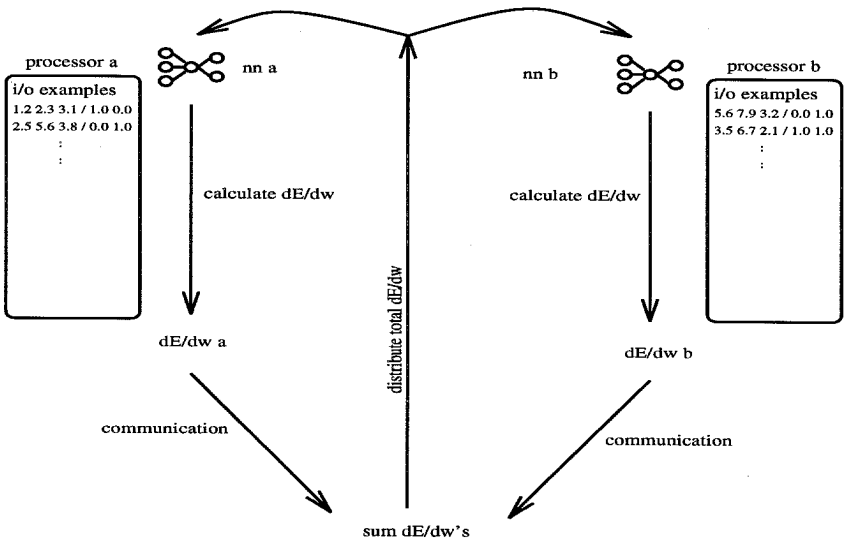

Figure 1.1: Multicomputer Training with Error Gradient Combination

different networks in each processor. Hence, the communication phase can no longer sum processor $\partial E/\partial w$'s. Instead, the communication phase must combine the different processor networks to produce a single network that combines the training from all processors (see Figure 1.2).

Output fusion and network combination need not be mutually exclusive learning methods. For example, our multicomputer training scheme could be altered to avoid network combination on the final communication phase. Instead, the algorithm could return all of the processors' final networks. Alternatively, the processors could be partitioned into blocks. Each block could train its own network using the network combination-based scheme. Then each block could contribute its final network to the output set of networks.

Network combination will be necessary for applications in which there is a critical shortage of space, processors, or time. If the outputs are to be combined, then the set of networks must be delivered to the performance site and stored there. If a combined network executes the task, then only the final network has to be delivered and stored – the others can be discarded after training and combination. In the first approach, every network in the set must be evaluated for every input. In the second approach, only the single combined network is evaluated for each input. Finally, for the set of networks, the ensemble of outputs must be combined for each input, making the set of networks slower than a single combined network even when there is an unlimited supply of storage and computational power.

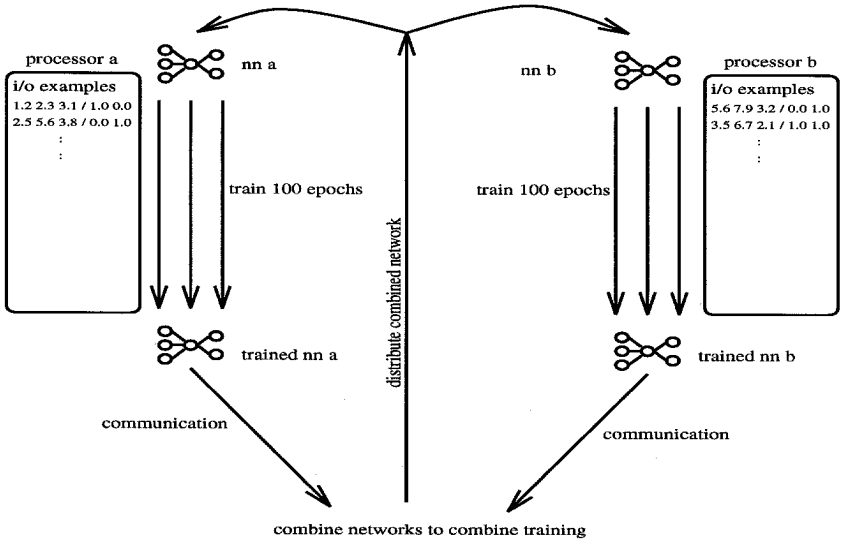

Figure 1.2: Multicomputer Training with Network Combination

Chapter 2

Architecture, Definitions, and Notation

This paper is concerned with feedforward networks that have full connections between successive layers and that have thresholded hyberbolic tangent sigmoids on all hidden and output units. Our combination algorithms operate on a pair of networks with identical architectures, and they produce a combined network with the same architecture. Define L to be the number of layers of weights in each network. Define n_h be the number of units fed by the weights in layer h, and define n_0 to be the number of input units. For each layer $h \in \{1 \ldots n\}$, each network's parameters consist of an $n_h \times n_{h-1}$ weight matrix W_h and an $n_h \times 1$ threshold vector $\underline{t}_h$. The weight matrices and threshold vectors are collectively referred to as the weights of a network.

The function of a network is defined:

$$nn(\underline{x}) \equiv \tanh(W_L(\ldots \tanh(W_1\underline{x} - \underline{t}_1)\ldots) - \underline{t}_L) \tag{2.1}$$

where $\underline{x}$ is an $n_0 \times 1$ input vector, and tanh is the componentwise hyberbolic tangent. For example, a 2 layer network executes the function:

$$\tanh(W_2(\tanh(W_1\underline{x} - \underline{t}_1)) - \underline{t}_2) \tag{2.2}$$

The networks to be matched will be referred to as nn^a and nn^b. The superscripts a and b will mark parts of nn^a and nn^b, respectively. For example, W_1^a refers to the weight vector for the first layer of nn^a. Also, unit j in layer h of a network will be denoted u_{hj}.

We assume the availability of a data set D, consisting of $|D|$ examples $(\underline{x},\underline{y})$, where $\underline{x}$ is an $n_0 \times 1$ input vector, and $\underline{y}$ is the corresponding $n_L \times 1$ target output vector. We refer to D as the matching data set, and we assume that its examples are drawn from the same distribution as the data for which the trained networks will be used.

Chapter 3

Pairwise Matching

If two single layer networks execute similar functions, then a weight-by-weight convex combination of the networks produces a network that executes a similar function. But multilayer network functions are invariant under several operations on the hidden units in each layer, so two networks may perform exactly the same function but have very different values of corresponding weights. Even if two multilayer networks execute exactly the same function, a weight-by-weight convex combination of the networks may execute a different function. Hence, a multilayer network convex combination algorithm must begin with a method to rearrange networks so that corresponding weights in the networks being combined perform corresponding roles in the networks' functions.

3.1 Permutations and Sign Flips

Since the hidden units in each layer of a network have symmetric roles in the network's function, network functions are invariant under permutations of the hidden units within any layer. To adjust the weights to reflect a permutation of the hidden units in layer h, first define the $n_h \times n_h$ permutation matrix P such that $p_{ij} = 1$ if the hidden unit in position i is to be moved to position j, and $p_{ij} = 0$ otherwise. Then make the assignments: $W_h := P^T W_h$, $\underline{t}_h := P^T \underline{t}_h$, and $W_{h+1} := W_{h+1} P$. This process permutes the rows of W_h, the elements of $\underline{t}_h$, and the columns of W_{h+1}.

Since the hyberbolic tangent is an odd function, the network function remains the same if the signs are changed on every weight into a hidden unit, the hidden unit's threshold, and every weight out of the hidden unit. We refer to this operation as a sign flip. To adjust the weights to reflect a sign flip on the hidden unit in position k of layer h, first define the $n_h \times n_h$ matrix C such that $c_{kk} = -1$, $c_{ii} = 1$ for all $i \neq k$, and $n_{ij} = 0$ for all $i \neq j$. Then make the assignments: $W_h := C W_h$, $\underline{t}_h := C \underline{t}_h$, and $W_{h+1} := W_{h+1} C$. This process

multiplies by -1 every weight in row k of W_h, position k of $\underline{t}_h$, and column k of W_{h+1}.

3.2 Canonical Form

Sussman [17] has shown that permutations and sign flips are the only invariances for irreducible 2 layer neural networks with hyperbolic tangent sigmoids, i.e. if two such networks execute the same function, then there is a sequence of permutations and sign flips that converts one network to the other. Chen, Lu, and Hecht-Nielson [4] present a method to convert a given network to a canonical form such that a pair of networks execute the same function if and only if their canonical forms agree weight by weight.

The following procedure converts a network to this canonical form. First, perform sign flips to make all thresholds nonnegative. Then permute each layer's hidden units so that their thresholds are in nondecreasing order.

Unfortunately, when this method is applied to pairs of networks trained on data sets drawn from the same distribution, it fails to match corresponding weights by function. So this procedure, which was developed to determine whether or not a pair of networks perform the same function, is not effective as a matching scheme for networks that execute similar functions.

Figure 3.1 illustrates a case in which this method fails as a matching algorithm. Weights in corresponding positions in the networks are either identical or almost identical. Combining the networks as presented produces a network with a similar function. The top network is in canonical form, but the bottom network is not. Converting the second network to canonical form flips the signs on the top hidden unit and swaps the center and bottom hidden units. Convex combinations of the networks in canonical form do not resemble either of the original networks.

In the next two sections we develop methods to match networks that have been trained on similar data. The methods perform well enough that there are weight-by-weight convex combinations of the matched networks that have less out-of-sample error than either of the original networks. Instead of converting both networks to canonical forms, these methods fix one network and match the other network to it by hidden unit permutations and sign flips (see Figure 3.2). These methods use the underlying operations of canonical form conversion, but they use different and more robust procedures for deciding how to apply permutations and sign flips.

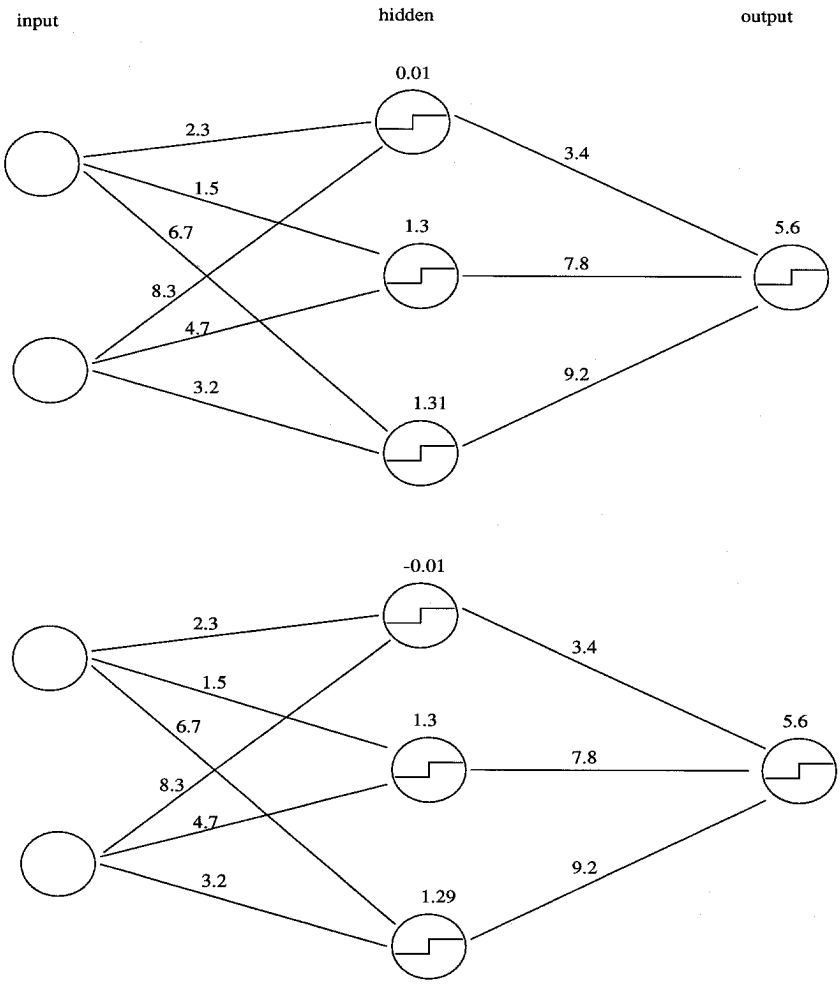

Figure 3.1: A Pair of Networks for which Canonical Form Matching Fails

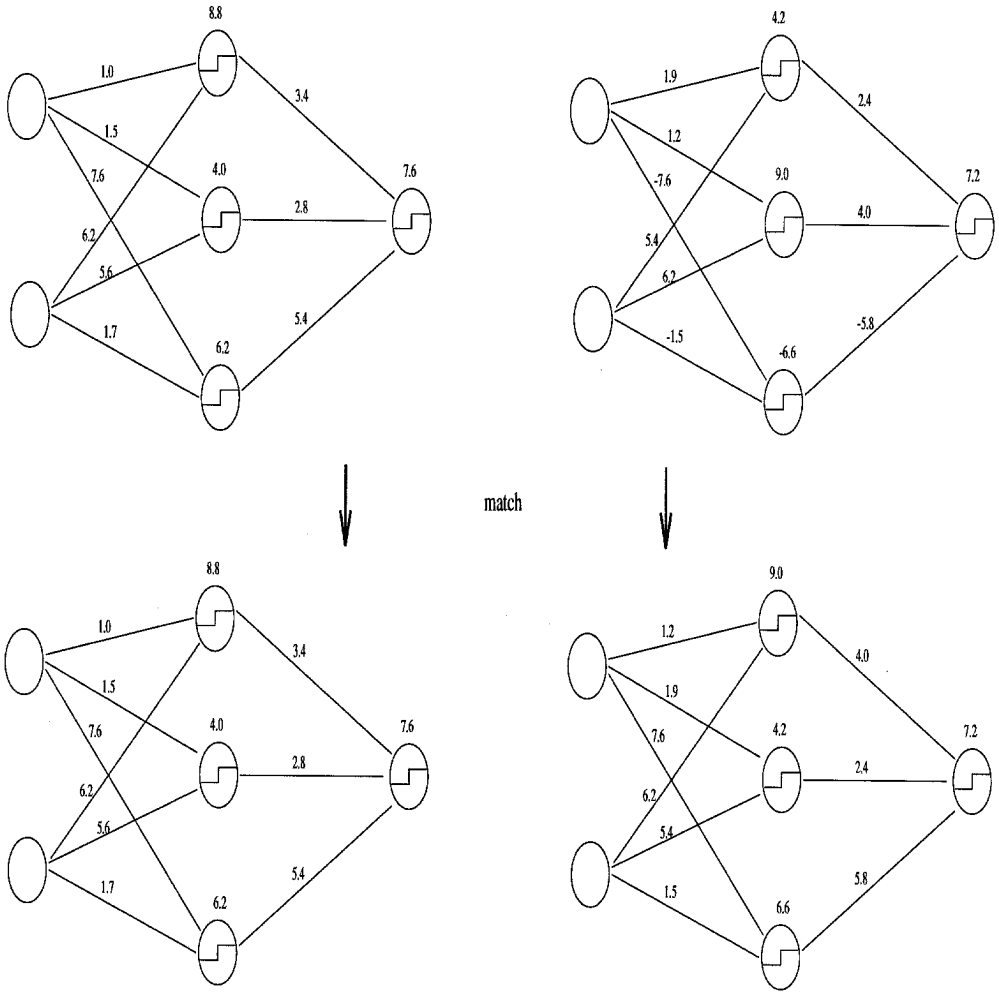

Figure 3.2: In HUF and BNF matching, one network is fixed and the other is matched to it by hidden unit permutations and sign flips. In this example, the network on the left is fixed, and the network on the right is adjusted – its top two hidden units are swapped, and the signs on its bottom hidden unit are flipped.

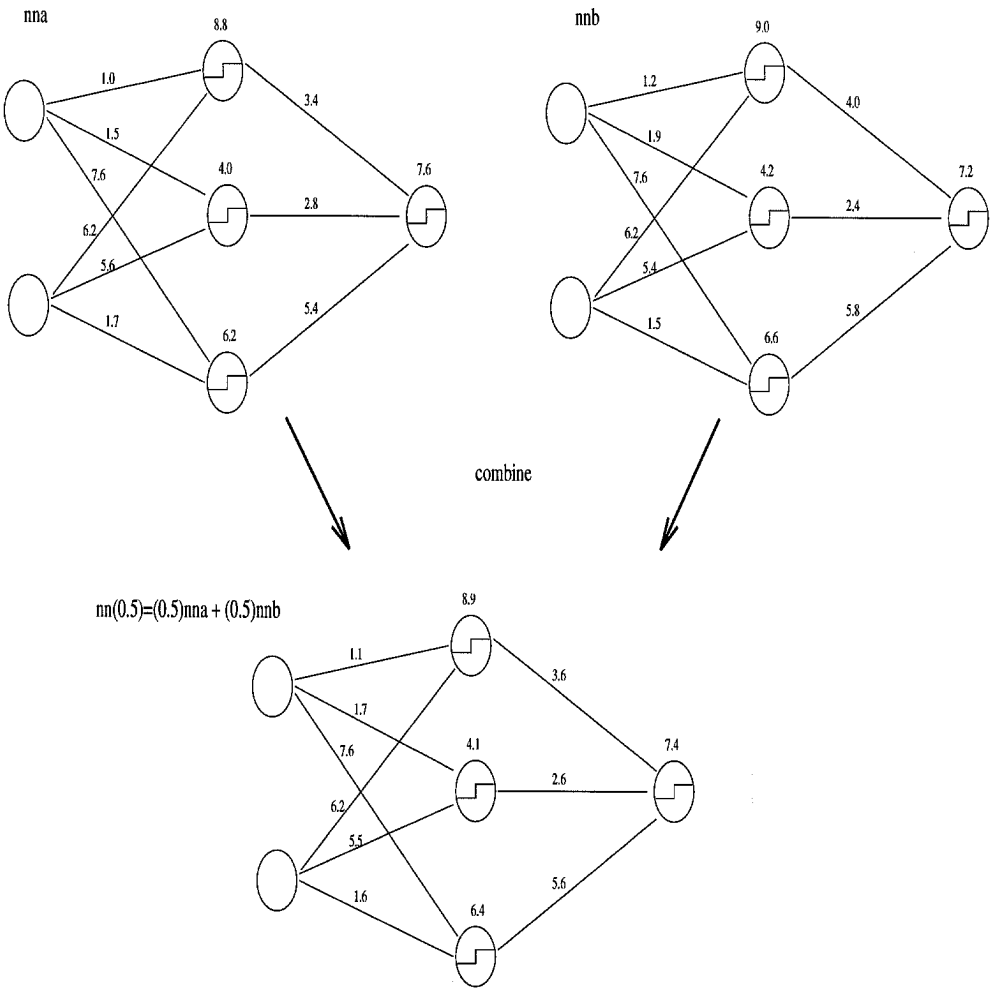

Figure 3.3: After successful matching, weight-by-weight convex combination produces a network that functions similarly to the networks being combined.

3.3 Hidden Unit Function (HUF) Algorithm

3.3.1 Overview

The hidden unit function is the function that determines the output of a hidden unit. The function of hidden unit i in layer h is:

$$u_{hi}(\underline{x}) \equiv [\tanh(W_h(\ldots \tanh(W_1\underline{x} - \underline{t}_1)\ldots) - \underline{t}_h)]_i \tag{3.1}$$

Shamir, Saad, and Marom have used hidden unit functions for pruning and genetic recombination [12, 13, 14].

The matching algorithm must identify functional correspondences among hidden units in the two networks being matched. Paired hidden units need not have functional agreement over the whole input space; it is more important for their functions to be similar in the regions of the domain from which the bulk of the inputs will be drawn in practice. Since the matching data set is drawn from the same distribution as the actual task inputs, the functional correspondence of hidden units over the matching data set inputs indicates the correspondence over the task inputs. The Hidden Unit Function (HUF) algorithm chooses a matching to maximize the similarity of paired hidden units' functions over the matching data set (see Figure 3.4).

3.3.2 Algorithm

HUF(nn^a, nn^b, D)
{
matrix S, F, P
real **similarity**, **flipped_similarity**
int h, i, j

$\forall\ h \in \{1 \ldots L-1\}$
  $\forall\ (i,j) \in \{1 \ldots n_h\} \times \{1 \ldots n_h\}$
    similarity $:= -\Sigma_{\underline{x} \in D}\ (u^a_{hi}(\underline{x}) - u^b_{hj}(\underline{x}))^2$
    flipped_similarity $:= -\Sigma_{\underline{x} \in D}\ (u^a_{hi}(\underline{x}) + u^b_{hj}(\underline{x}))^2$
    $s_{ij} :=$ max(**similarity**,**flipped_similarity**)
    $f_{ij} := 0$ if **similarity** > **flipped_similarity**; $f_{ij} := 1$ otherwise
  $P :=$ permutation matrix that maximizes $\Sigma_{ij} s_{ij} p_{ij}$
  $\forall\ (i,j) \in \{1 \ldots n_h\} \times \{1 \ldots n_h\}$
    if $p_{ij} = 1$ and $f_{ij} = 1$ then flip the signs on hidden unit u^a_{hi}
  permute layer h of nn^a by P
return(nn^a,nn^b)
}

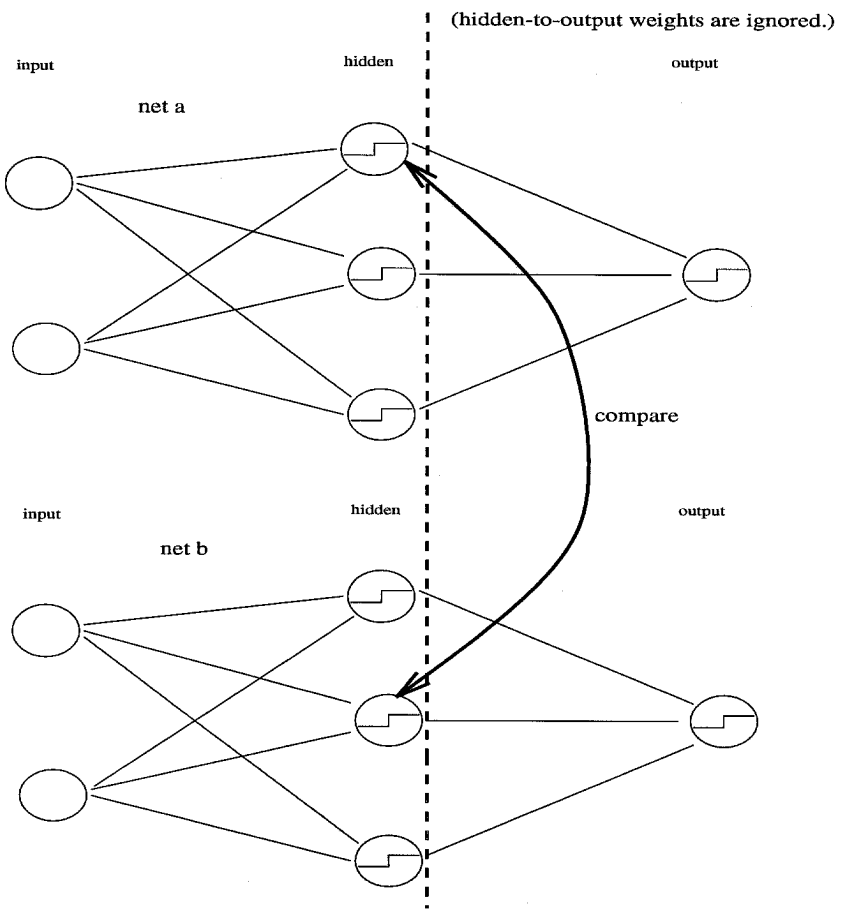

Figure 3.4: Determining Hidden Unit Similarity in the HUF Algorithm

The algorithm pairs hidden units layer by layer. In each layer, s_{ij} is a measure of the similarity between hidden unit i in nn^a and hidden unit j in nn^b. The variable **similarity** is assigned the negative of the sum of squared differences of the hidden unit functions over the inputs in the matching data set. To account for the possibility of sign flipping, the variable **flipped_similarity** is assigned the value that would be assigned to **similarity** if one of the hidden unit's signs were flipped. The overall similarity s_{ij} is the maximum of **similarity** and **flipped_similarity**.

The permutation that gives a best matching corresponds to an independent set of elements in S with maximum sum. Therefore, the permutation matrix P can be found by applying Kuhn's Hungarian algorithm [5, 7] to the matrix $-S$. The algorithm returns a set of element positions containing one position in each row and one position in each column. To produce the permutation matrix P, set the elements in these positions to 1, and set all other elements to 0.

The matrix F records the pairs of hidden units for which a sign flip improves the similarity. For each such pair included in the matching induced by P, the signs are flipped on one of the pair's hidden units.

Note that the HUF algorithm does not use the outputs of the matching data set examples, i.e. the algorithm never accesses $\underline{y} \in D$. Hence, the matching data set outputs need not be known. If the input distribution is known, then inputs $\underline{x}$ can be generated at random to enhance the matching data set. So the accuracy of the matching need not be limited by a scarcity of in-sample data [2].

3.3.3 Complexity

Each evaluation of $u_{hi}(\underline{x})$ requires $O(Ln^2)$ time, where $n \equiv \max\{n_0 \ldots n_L\}$, since the evaluation consists of multiplying the matrices $W_1 \ldots W_L$ by vectors. There are $O(Ln^2|D|)$ evaluations in the algorithm, so it has time complexity $O(L^2n^4|D|)$. The Hungarian algorithm has time complexity $O(n^4)$, so it does not affect the time complexity of the HUF algorithm. That is, the time required to compute the similarity matrix S dominates the time required to find the best matching in S. Since the algorithm requires space to store the matrices S, F, and P, it has space complexity $O(n^2)$.

The time complexity can be decreased by using more space. If $O(n|D|)$ space is availablethen, for each layer h, the values $u^a_{hi}(\underline{x})$ and $u^b_{hi}(\underline{x})$ can be precomputed for all $(i, \underline{x}) \in \{1 \ldots n_h\} \times D$. These values can then be accessed as needed instead of re-evaluating the hidden unit functions every time their values are needed in the computation of S. For each $\underline{x} \in D$, a single pass through the network inputs to layer h computes $u^a_{hi}(\underline{x})$ for all hidden units in the layer. Thus, precomputing all $u^a_{hi}(\underline{x})$ for a single layer requires $O(Ln^2|D|)$ time. In each layer, the computation of S requires $O(n^2|D|)$ time, and the Hungarian algorithm requires $O(n^4)$ time. So the total time complexity is $O(L(Ln^2|D| + n^2|D| + n^4))$ $= O(L^2n^2|D| + Ln^4)$.

If $O(Ln|D|)$ space is available, then all hidden unit function values for all layers can be precomputed together, saving even more time. For each $\underline{x} \in D$, all hidden unit functions $u^a_{hi}(\underline{x})$ can be evaluated with a single pass through the network. So the entire precomputation requires $O(Ln^2|D|)$ time. Thus, the algorithm has time complexity $O(Ln^2|D| + L(n^2|D| + n^4)) = O(Ln^2|D| + Ln^4)$. Assuming that the size of the data set is on the order of the number of weights in each network, i.e. $O(|D|) = O(Ln^2)$, the algorithm has time complexity $O(Ln^4)$. In this case, the time required to precompute the hidden unit functions and to compute the similarity matrix dominates the time required to find the best matching.

Here are a few facts to give the reader some perspective on time complexity as it relates to network algorithms. There are $O(Ln^2)$ weights in each network. Network function evaluation (a forward pass through the weights) requires $O(Ln^2)$ time. Backpropogation (a backward pass) also requires $O(Ln^2)$ time. Each epoch of training on data set D requires $O(Ln^2|D|)$ time. Evaluation of a network's error over data set D also requires $O(Ln^2|D|)$ time.

For example, in the test problems examined later in this paper, 10-10-10 networks are trained on data sets with twice as many examples as the number of weights in the networks. The networks have about 200 weights ($Ln^2 \approx 200$), and the data sets have about 400 examples ($|D| \approx 400$.) Each network function evaluation requires about 200 flops, and an epoch of training or an error evaluation requires about 80,000 flops. If there is no limit on space, then matching a pair of networks by the HUF algorithm requires about 100,000 flops – 40,000 for each network to pass all inputs through to the hidden layer and about 10,000 to find the best matching in the similarity matrix.

3.4 Bottleneck Function (BNF) Algorithm

3.4.1 Overview

When the HUF algorithm compares a pair of hidden units, it ignores the layers of the networks beyond the layer in which the hidden units occur. The decision to match a pair of hidden units is based only on the weights between the input units and the pair of hidden units. The units' "downstream" effects are ignored. The bottleneck function algorithm uses both the "downstream" and the "upstream" portions of the networks to match hidden units.

The bottleneck network of hidden unit u_{hi}, denoted by nn_{hi}, is formed by removing from nn all units in layer h except the unit in position i. This is equivalent to zeroing all elements of W_h, $\underline{t}_h$, and W_{h+1} except row i of W_h, element i of $\underline{t}_h$, and column i of W_{h+1}. The bottleneck algorithm matches hidden units to maximize the similarity over the matching data set of the functions of paired hidden units' bottleneck networks (see Figure 3.5).

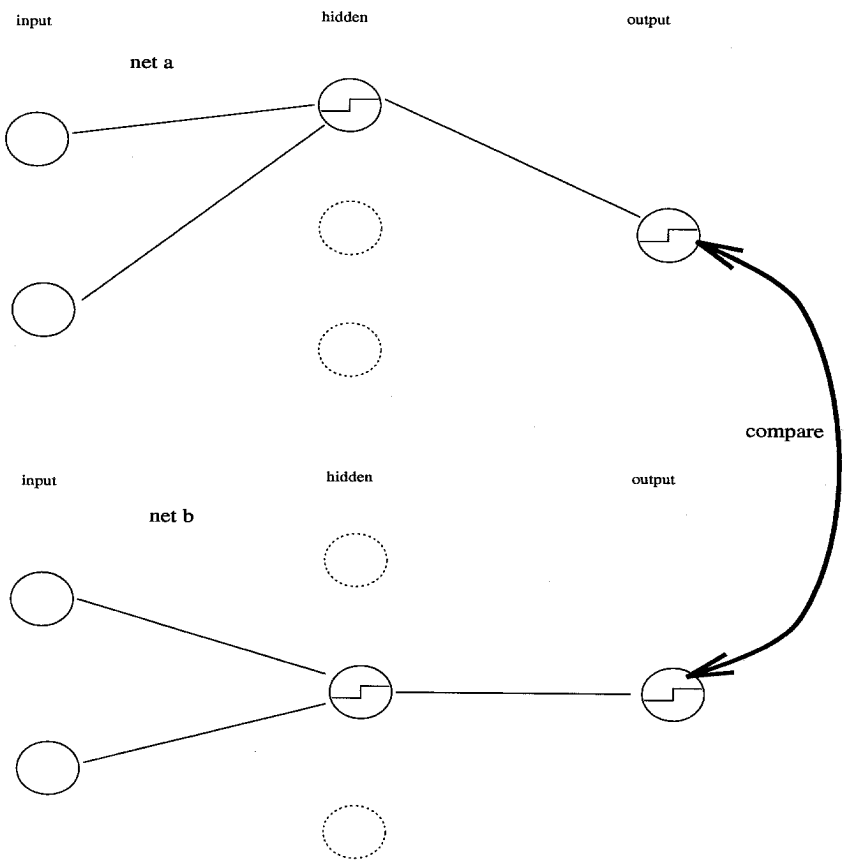

Figure 3.5: Determining Hidden Unit Similarity in the BNF Algorithm

3.4.2 Algorithm

BNF(nn^a, nn^b, D)
{
matrix S, P
int h, i, j

$\forall\, h \in \{1 \ldots L-1\}$
  $\forall\, (i,j) \in \{1 \ldots n_h\} \times \{1 \ldots n_h\}$
    $s_{ij} := -\Sigma_{\underline{x} \in D} ||nn^a_{hi}(\underline{x}) - nn^b_{hj}(\underline{x})||^2$
  $P :=$ permutation matrix that maximizes $\Sigma_{ij} s_{ij} p_{ij}$
  permute layer h of nn^a by P
  $\forall\, j \in \{1 \ldots n_h\}$
    if correlation(u^a_{hj},u^b_{hj}) < 0 then flip the signs on u^a_{hj}
return(nn^a,nn^b)
}

The bottleneck function algorithm has the same basic structure as the hidden unit function algorithm. For each layer, a similarity matrix is computed, and its maximal independent set determines the permutation used to pair hidden units between the networks. The BNF algorithm determines sign flips differently than the HUF algorithm. Since the bottleneck network function (like the original network function) is invariant under sign flips, they are determined independently from the permutation. For each pair of matched hidden units, the correlation is computed over all weights incident on the hidden units as follows.

$$\text{correlation}(u^a_{hj},u^b_{hj}) \equiv \frac{1}{n_{h-1}+1+n_{h+1}} (\Sigma_{i=1}^{n_{h-1}} [W^a_h]_{ji} [W^b_h]_{ji} + [t^a_h]_j [t^b_h]_j + \Sigma_{k=1}^{n_{h+1}} [W^a_{h+1}]_{kj} [W^b_{h+1}]_{kj})$$

If the hidden units' weights are negatively correlated, then the signs are flipped on one of the two hidden units, producing a positive correlation.

As in the HUF algorithm, the matching data set outputs $\underline{y} \in D$ are not used by the BNF algorithm. Thus, if the input distribution is known, then the matching data can be enhanced with randomly drawn inputs [2].

3.4.3 Complexity

Each bottleneck network function evaluation requires $O(Ln^2)$ time. The algorithm executes $O(Ln^2|D|)$ evaluations, so it has time complexity $O(L^2n^4|D|)$. Space is required to store matrices S and P, so the algorithm has space complexity $O(n^2)$.

As with the HUF algorithm, the time complexity of the BNF algorithm can be decreased by using more space. Precomputing all bottleneck function evaluations in each layer requires $O(n^2|D|)$ space since each bottleneck function

returns an $n_L \times 1$ vector, and each network has $O(n)$ bottleneck networks for each layer – one for each hidden unit in the layer. For each layer, the precomputation requires $O(Ln^3|D|)$ time; the computation of S requires $O(n^3|D|)$ time, and the Hungarian algorithm requires $O(n^4)$ time. Hence, the time complexity for each layer is $O(Ln^3|D| + n^4)$. The time complexity for the entire algorithm is $O(L^2n^3|D| + Ln^4)$. Assuming that the number of data points is on the order of the number of weights in each network, the time required for precomputation dominates the time required for the Hungarian algorithm, so the BNF algorithm has time complexity $O(L^2n^3|D|)$.

Chapter 4

Pairwise Proportioning

We refer to choosing a convex combination of matched networks as proportioning. Define $nn(\theta) \equiv (1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b$, where $n\hat{n}^a$ and $n\hat{n}^b$ are matched networks. Ideally, in pairwise combination, proportioning chooses the value of θ that minimizes the test error of $nn(\theta)$. We assume that there is some data set D_p available for proportioning; it may be either "fresh" in-sample data or data previously used for training and matching.

4.1 Grid Search

A simple, robust proportioning method is grid search, i.e. choose the combination $nn(\theta)$ that minimizes error on the proportioning data set over $\theta \in \{0, \triangle\theta, 2\triangle\theta, \ldots, 1\}$. Grid search requires $\frac{1}{\triangle\theta} + 1$ error evaluations. Smaller grid spacing $\triangle\theta$ requires more computation, but it gives greater accuracy in identifying the network that minimizes error on the proportioning data set.

4.2 Binary Search

More sophisticated methods can be used to minimize the error of the convex combination network on the proportioning data. For example, binary search [6] can be used to find local minima of the error by finding θ such that $\frac{\partial E}{\partial \theta} = 0$. The quantity $\frac{\partial E}{\partial \theta}$ can be computed using the chain rule.

$$\frac{\partial E}{\partial \theta} = \frac{\partial E}{\partial nn(\theta)} \cdot \frac{\partial nn(\theta)}{\partial \theta} \tag{4.1}$$

The first term on the RHS is the gradient of the error with respect to the weights of a network, which can be computed by backpropogation [11]. To evaluate the

second term, recall that $nn(\theta) \equiv (1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b$.

$$\frac{\partial nn(\theta)}{\partial \theta} = \frac{\partial}{\partial \theta}[(1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b] \tag{4.2}$$

$$= n\hat{n}^b - n\hat{n}^a \tag{4.3}$$

Hence, the quantity $\frac{\partial E}{\partial \theta}$ can be computed by backpropogation followed by a dot product.

$$\frac{\partial E}{\partial \theta} = \frac{\partial E}{\partial nn(\theta)} \cdot (n\hat{n}^b - n\hat{n}^a) \tag{4.4}$$

4.3 Hybrid Proportioning

The endpoints of the binary search can be set to $\theta = 0$ on the left and $\theta = 1$ on the right, but a more robust search results from combining a coarse grid search to identify regions of $[0, 1]$ containing local minima with a precise binary search for a minimum in each of the regions. We refer to this process as hybrid proportioning.

The grid search examines $\theta \in [0, 1]$ at regular intervals to find regions with error decreasing on the left and increasing on the right. (These regions contain local minima.) So the grid search evaluates $\frac{\partial E}{\partial \theta}$ for each $\theta \in \{0, \triangle\theta, 2\triangle\theta, \ldots, 1\}$. The regions $[k\triangle\theta, (k+1)\triangle\theta]$ with $\frac{\partial E}{\partial \theta} \leq 0$ at $\theta = k\triangle\theta$ and $\frac{\partial E}{\partial \theta} \geq 0$ at $\theta = (k+1)\triangle\theta$ contain local minima. A binary search is performed in each such region to find a local minimum. For each regional local minimum θ_k, the error of $nn(\theta_k)$ is evaluated over the proportioning data. A value of θ_k that gives the least error is returned as the result of proportioning.

The following procedures perform hybrid proportioning. The binary search procedure explores $\theta \in [\theta_l, \theta_r]$ in search of a local minimum where $\frac{\partial E}{\partial \theta} = 0$. The procedure is built on the assumptions that $\frac{\partial E}{\partial \theta} \leq 0$ at θ_l and $\frac{\partial E}{\partial \theta} \geq 0$ at θ_r. The value L is the number of levels of recursion.

binary(θ_l,θ_r,L)
{
$\theta_c := \frac{\theta_l + \theta_r}{2}$
if $\frac{\partial E}{\partial \theta} \leq 0$ at θ_c
    then return binary(θ_c,θ_r,$L-1$)
    else return binary(θ_l,θ_c,$L-1$)
}

The hybrid proportioning procedure has two parameters – the grid spacing $\triangle\theta$ and the number of levels L in each binary search.

hybrid($\triangle\theta$,L)
{
for $k := 0$ to $\frac{1}{\triangle\theta} - 1$
    if $\frac{\partial E}{\partial \theta} \leq 0$ at $k\triangle\theta$ and $\frac{\partial E}{\partial \theta} \geq 0$ at $(k+1)\triangle\theta$
        then $\theta_k :=$ binary($k\triangle\theta$,$(k+1)\triangle\theta$,L)
        else $\theta_k := 0$
return θ_k that minimizes the error of $nn(\theta_k)$ over $k \in \{0 \ldots \frac{1}{\triangle\theta} - 1\}$
}

The hybrid method is a good blend of robustness and efficiency. The method requires $\frac{1}{\triangle\theta} + 1$ backpropogation passes to evaluate $\frac{\partial E}{\partial \theta}$ at the grid points, up to $L\frac{1}{\triangle\theta}$ backpropogation passes to evaluate $\frac{\partial E}{\partial \theta}$ in the binary searches, and up to $\frac{1}{\triangle\theta}$ error evaluations to choose the best convex combination $nn(\theta_k)$. This makes $(L+1)(\frac{1}{\triangle\theta} + 1) + 1$ backpropogation passes and $\frac{1}{\triangle\theta}$ error evaluations. The method works well in practice.

4.4 Newton's Method

For further sophistication and accuracy, the binary search could be replaced by Newton's method. The goal is to find a root of $f(\theta) = \frac{\partial E}{\partial \theta}$, so the iteration has the following form.

$$\theta^{k+1} = \theta^k - \frac{f(\theta^k)}{f'(\theta^k)} \tag{4.5}$$

$$\theta^{k+1} = \theta^k - \frac{\frac{\partial E}{\partial \theta}\big|_{\theta=\theta^k}}{\frac{\partial^2 E}{\partial \theta^2}\big|_{\theta=\theta^k}} \tag{4.6}$$

The second derivative could either be computed directly or estimated by a finite difference approximation:

$$\left.\frac{\partial^2 E}{\partial \theta^2}\right|_{\theta=\theta^k} \approx \frac{\frac{\partial E}{\partial \theta}\big|_{\theta=\theta^k+\triangle\theta} - \frac{\partial E}{\partial \theta}\big|_{\theta=\theta^k-\triangle\theta}}{2\triangle\theta} \tag{4.7}$$

$$\approx \frac{1}{2\triangle\theta}\left[\frac{\partial E}{\partial nn(\theta^k+\triangle\theta)} \cdot (\hat{nn^b} - \hat{nn^a}) - \frac{\partial E}{\partial nn(\theta^k-\triangle\theta)} \cdot (\hat{nn^b} - \hat{nn^a})\right] \tag{4.8}$$

$$\approx \frac{1}{2\triangle\theta}\left[\frac{\partial E}{\partial nn(\theta^k+\triangle\theta)} - \frac{\partial E}{\partial nn(\theta^k-\triangle\theta)}\right] \cdot (\hat{nn^b} - \hat{nn^a}) \tag{4.9}$$

Newton's method requires more computation per iteration than binary search, but it has a higher rate of convergence.

4.5 A Fast Heuristic Method

An extremely fast and simple proportioning method is to choose $\hat{\theta} = E^a/(E^a + E^b)$, where E^a and E^b are the errors of nn^a and nn^b on the proportioning data set. The formula is based on the observation that the lowest error combination generally lies closer to the student network with lower error. This method requires only 2 error evaluations, and it works well in practice.

Chapter 5

Pairwise Matching and Combination Tests

The results of two tests are presented in this section. The first test compares combinations of trained networks after HUF, BNF, and canonical form matching to combinations without matching. A pair of networks are trained on separate data sets, and the training is interrupted at intervals to measure the test errors of the networks' convex combinations under the different matching schemes.

The second test examines network combination at work. As before, there are two sets of training data. Two networks are initialized with random weights. The networks are matched, and a convex combination of the networks is chosen. The combination network is duplicated, and one of its copies trains on each data set. Through training, the networks drift apart in weight space. After an interval, the training is interrupted, and the process of matching, combining, duplicating, and training is repeated. This process models the multicomputer training scheme with data parallelism that was discussed earlier in this paper. The test uses HUF matching.

Both tests employ the twin networks framework [1]. Weights are drawn at random to form a teacher network. The teacher network's function is the target function. To create data, inputs are drawn at random and fed to the teacher network to create target outputs. Then the teacher network is destroyed, and randomly initialized student networks with the same architecture as the teacher net train on the data.

Each test run has its own function, represented by a randomly generated teacher net. Four data sets are used – training sets D_a and D_b, matching set D_m, and test set D_t. The student networks are nn^a and nn^b. Each test run consists of 100 intervals. Each interval consists of a matching phase and a training phase. In each training phase, each student network trains for 20 epochs. Each epoch consists of sequential mode backpropogation, with step size

0.01.

All of the networks in the tests have 2 layers, with 10 input units, 10 hidden units, and 10 output units. The teacher network's weights are drawn independently and uniformly at random from $[-10, 10]$. The student network weights are drawn from $[-0.1, 0.1]$. The data set inputs were drawn from $[-1, 1]^{10}$. Each data set has twice as many examples as the number of weights in each network.

The error of network nn on data set D is defined:

$$E(nn, D) \equiv \frac{1}{2|D|n_L} \Sigma_{(\underline{x},\underline{y})\in D} ||nn(\underline{x}) - \underline{y}||^2 \tag{5.1}$$

where n_L is the number of output units.

The network formed by a weight-by-weight convex combination of nn^a and nn^b is denoted by $(1-\theta)nn^a + (\theta)nn^b$.

5.1 Matching Methods Compared

Each test run follows the procedure:

1. Randomly generate a teacher network.
2. Apply the teacher network to random inputs to create data sets D_a, D_b, D_m, and D_t.
3. Randomly initialize nn^a and nn^b.
4. Interval (repeat 100 times):
   a. Match and Output
      i. Straightforward Combination
         Show E$((1-\theta)nn^a + (\theta)nn^b, D_t)$ $\forall\ \theta \in \{0.00, 0.05, \ldots, 1.00\}$
      ii. Canonical Form Matching
         $n\hat{n}^a :=$ canonical_form(nn^a), $n\hat{n}^b :=$ canonical_form(nn^b)
         Show E$((1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b, D_t)$ $\forall\ \theta \in \{0.00, 0.05, \ldots, 1.00\}$
      iii. HUF Matching
         $(n\hat{n}^a, n\hat{n}^b) :=$ HUF(nn^a, nn^b, D_m)
         Show E$((1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b, D_t)$ $\forall\ \theta \in \{0.00, 0.05, \ldots, 1.00\}$
      iv. BNF Matching
         $(n\hat{n}^a, n\hat{n}^b) :=$ BNF(nn^a, nn^b, D_m)
         Show E$((1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b, D_t)$ $\forall\ \theta \in \{0.00, 0.05, \ldots, 1.00\}$
   b. Train
      i. train nn^a on D_a for 20 epochs
      ii. train nn^b on D_b for 20 epochs

In Figures 5.1, 5.2, 5.3, and 5.5, each curve shows the test errors of the convex combinations of the student networks – either without matching or after matching by canonical form, HUF, or BNF. The endpoints of each curve are the test errors of the student networks. Since test errors generally decreased during

training (see Figures 5.13 and 5.14), the endpoints generally proceed down the figures' sides with each interval. If a curve contains a point that is lower than either endpoint, then there is a convex combination that has lower test error than either of the combined networks, so the matching method is successful.

Note the curves at the top of Figures 5.1, 5.2, 5.3, and 5.5, which display the test errors of convex combinations of the student networks before any training occurs. Straightforward combination and combination after each matching scheme all perform about as well. The curve is quite linear, possibly because the student networks are initialized with small weights, so their sigmoids are initially in their linear regions. Hence, the student networks are initially nearly linear.

The curves for both straightforward combination (Figure 5.1) and combination with canonical form matching (Figure 5.2) are both concave down, indicating that there are no combination networks that improve on the student networks. After the first interval of training, the curves for combination after HUF matching (Figure 5.3) and for combination after BNF matching (Figure 5.5) are both concave up. In the early intervals, the curves are almost linear. Early in training, the best convex combination of the student networks has a higher test error than either of the student networks have after the next interval of training. So combination would not make a big contribution to the learning process. Now observe Figures 5.4 and 5.6, which show the curves for combination after HUF and BNF matching late in training. The lowest point on each curve is below the endpoints of the next lower curve, indicating that the error could be decreased more by combining the students than by training for another interval. As training progresses, matching and taking the best convex combination reduces the error as much as training for many intervals.

Note that combination networks are constructed only to observe their errors, then they are discarded. Hence, each curve shows the effects of training nn_a and nn_b separately, then combining the networks. For example, the curve with the lowest endpoints in each figure illustrates the results of independently training a pair of networks on separate data sets for 2000 epochs each, then combining the networks.

Figures 5.7 and 5.8 show the average results for 50 tests like the one used to generate Figures 5.1, 5.2, 5.3, and 5.5. The similarity to the earlier figures indicates that the single run illustrated in the earlier figures is typical. Figure 5.8 focuses on the later intervals of training. The curves' double dip shape is an artifact of averaging over curves that each have a single dip. The double dip indicates that the lowest error convex combination network is typically not a balanced combination of the student networks. In general, the best combination lies closer to the student network with lower error. (This can be observed in Figures 5.4 and 5.6.) Note that in Figure 5.8, the HUF and BNF curves are different. Hence, HUF and BNF do not always produce the same matching.

To explore the dependence of the matching methods on both training sets being drawn from the same distribution, a test was run in which the train-

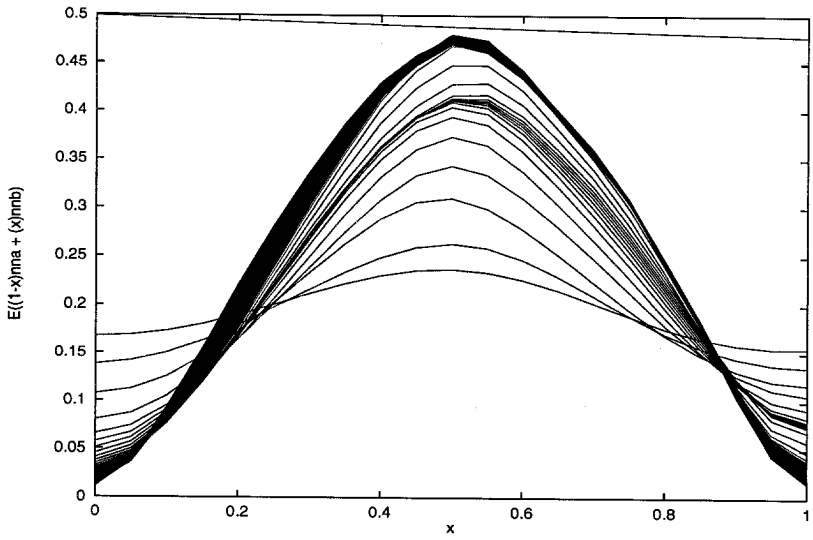

Figure 5.1: Test Errors of Convex Combinations Without Matching

ing data set inputs were drawn uniformly at random from separate halves of the input space $[-1,1]^{10}$. The inputs for D_a are drawn uniformly at random from $[-1,0] \times [-1,1]^9$, and the inputs for D_b are drawn uniformly at random from $[0,1] \times [-1,1]^9$. The test and matching set inputs are drawn uniformly at random from the whole space. The results with nonhomogeneous training data are similar to those with homogeneous training data. With nonhomogeneous training data, once again combination without matching and combination with canonical form matching failed to produce networks with lower test errors than the trained networks. Combination after HUF and BNF matching is a bit weaker with nonhomogeneous data. In the first few training epochs, HUF and BNF matchings sometimes produce combination networks with higher test error than the networks being combined. However, after a few hundred training epochs, combination produces networks with lower test errors than the networks being combined. Thus, the later error curves are quite similar to those for the test with homogeneous data. The curves showing the average combination network test errors over 50 runs is almost identical to Figure 5.7, which shows results for the same test with homogeneous data.

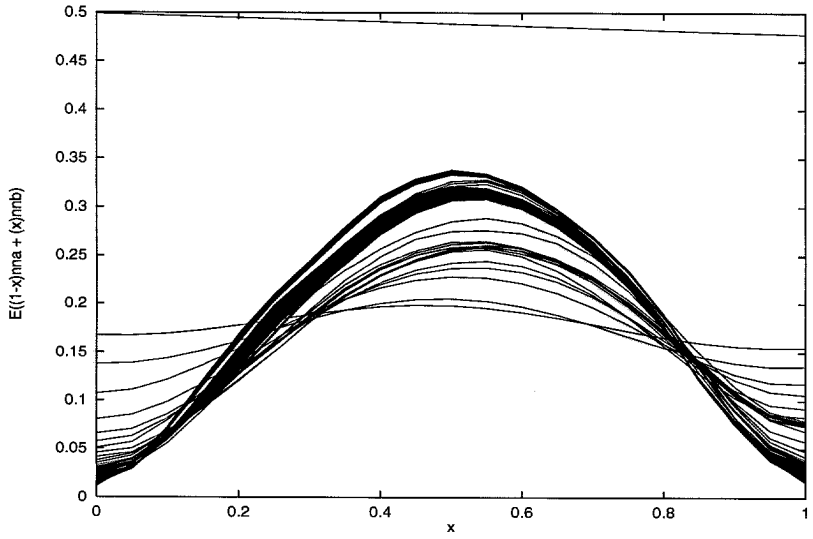

Figure 5.2: Test Errors of Convex Combinations With Canonical Form Matching

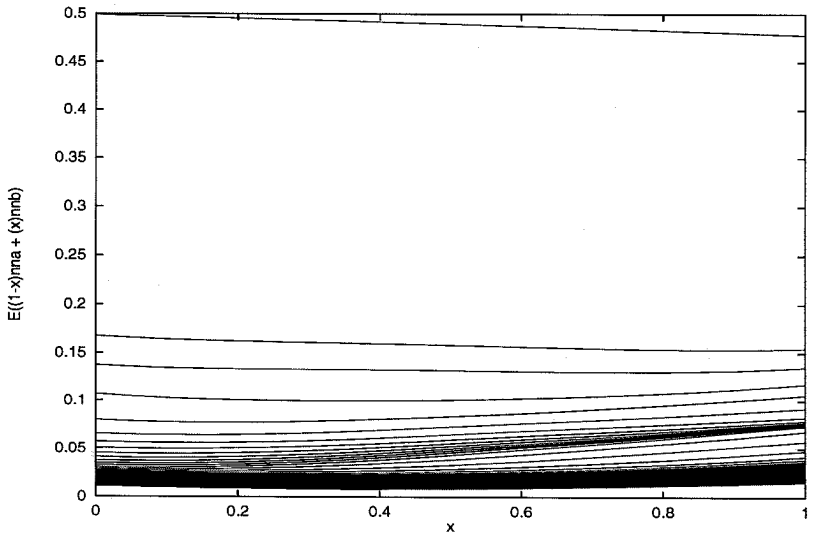

Figure 5.3: Test Errors of Convex Combinations With Hidden Unit Function (HUF) Matching

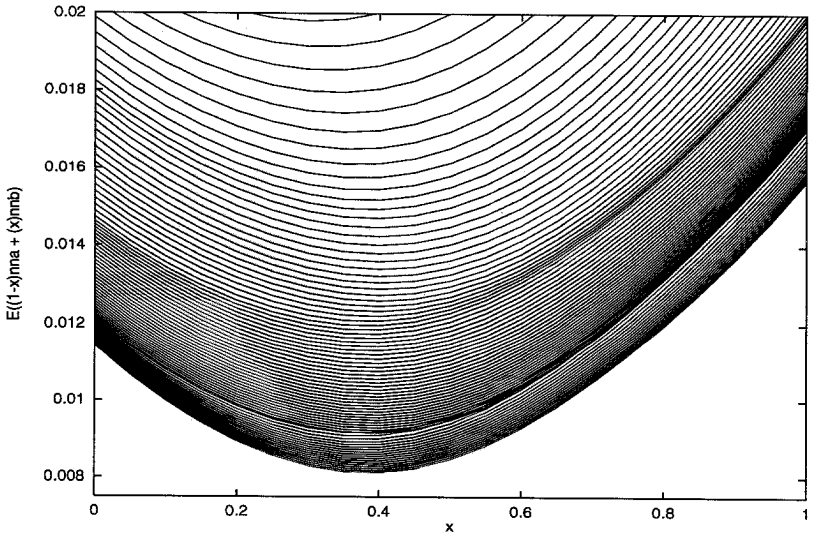

Figure 5.4: Hidden Unit Function (HUF) Matching Performance After Many Epochs of Training (Detail of Figure 5.3)

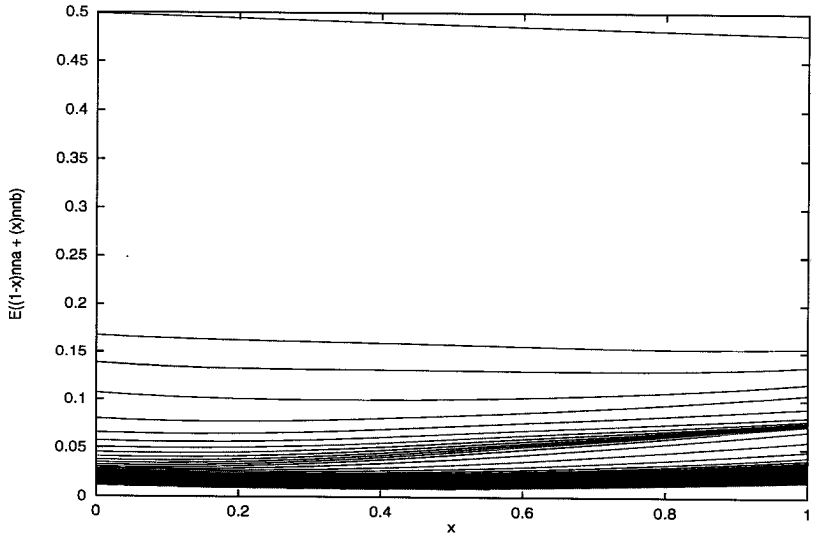

Figure 5.5: Test Errors of Convex Combinations With Bottleneck Function (BNF) Matching

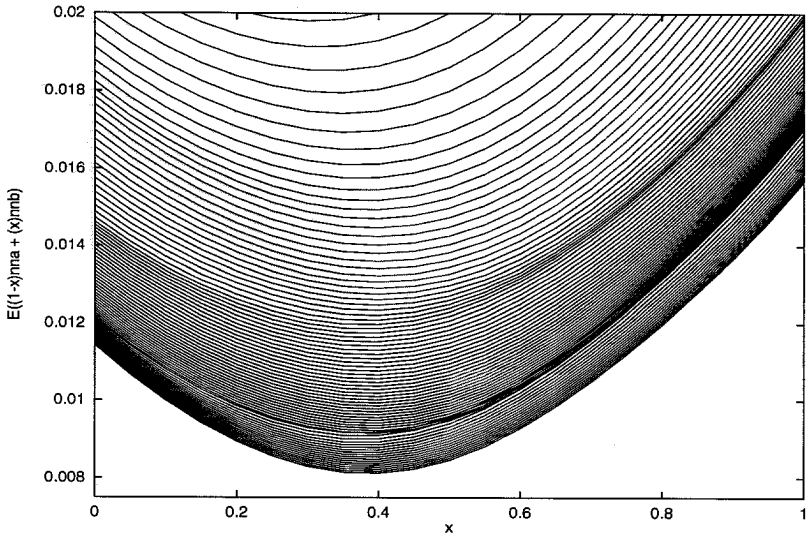

Figure 5.6: Bottleneck Function (BNF) Matching Performance After Many Epochs of Training (Detail of Figure 5.5)

5.2 Combination at Work

In the previous test, combination networks were formed only to observe their errors, then discarded. In this test, the combination networks are trained and recombined. Hence, combination plays a role in training. After each training interval, the student networks are matched, proportioned, and replaced by a convex combination of the matched students.

Each test run follows the procedure:

1. Randomly generate a teacher network.
2. Apply the teacher network to random inputs to create data sets D_a, D_b, D_m, and D_t.
3. Randomly initialize nn^a and nn^b.
4. Interval (repeat 100 times):
   a. Match: $(n\hat{n}^a, n\hat{n}^b) := \text{HUF}(nn^a, nn^b, D_m)$
   b. Output
      i. Test Error: Show $\text{E}((1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b, D_t)\ \forall\ \theta \in \{0.00, 0.05, \ldots, 1.00\}$
      ii. Match Error: Show $\text{E}((1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b, D_m)\ \forall\ \theta \in \{0.00, 0.05, \ldots, 1.00\}$
   c. Proportion: $\hat{\theta} := \theta \in \{0.00, 0.05, \ldots, 1.00\}$ that minimizes $\text{E}((1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b, D_m)$
   d. Duplicate
      i. $nn^a := (1-\hat{\theta})n\hat{n}^a + (\hat{\theta})n\hat{n}^b$

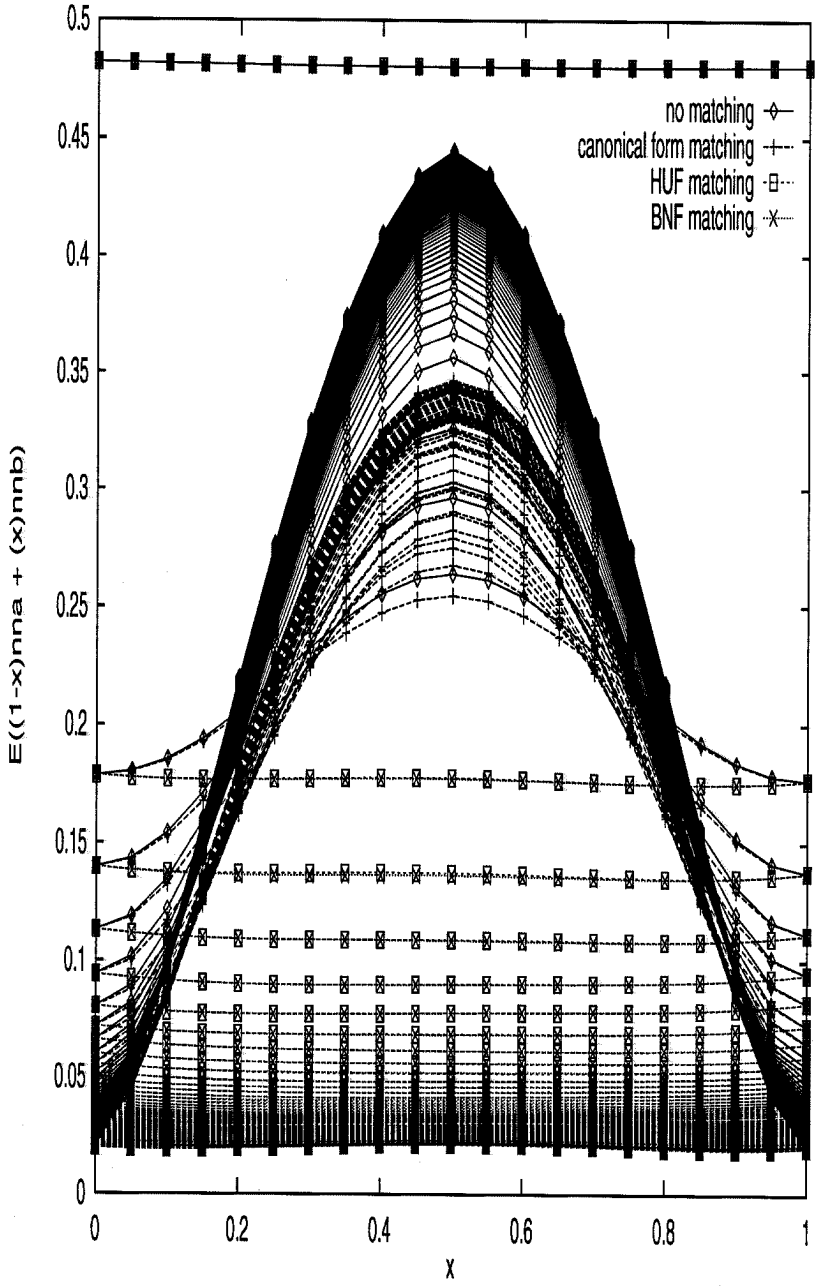

Figure 5.7: Comparison of Matching Methods, Average Over 50 Runs

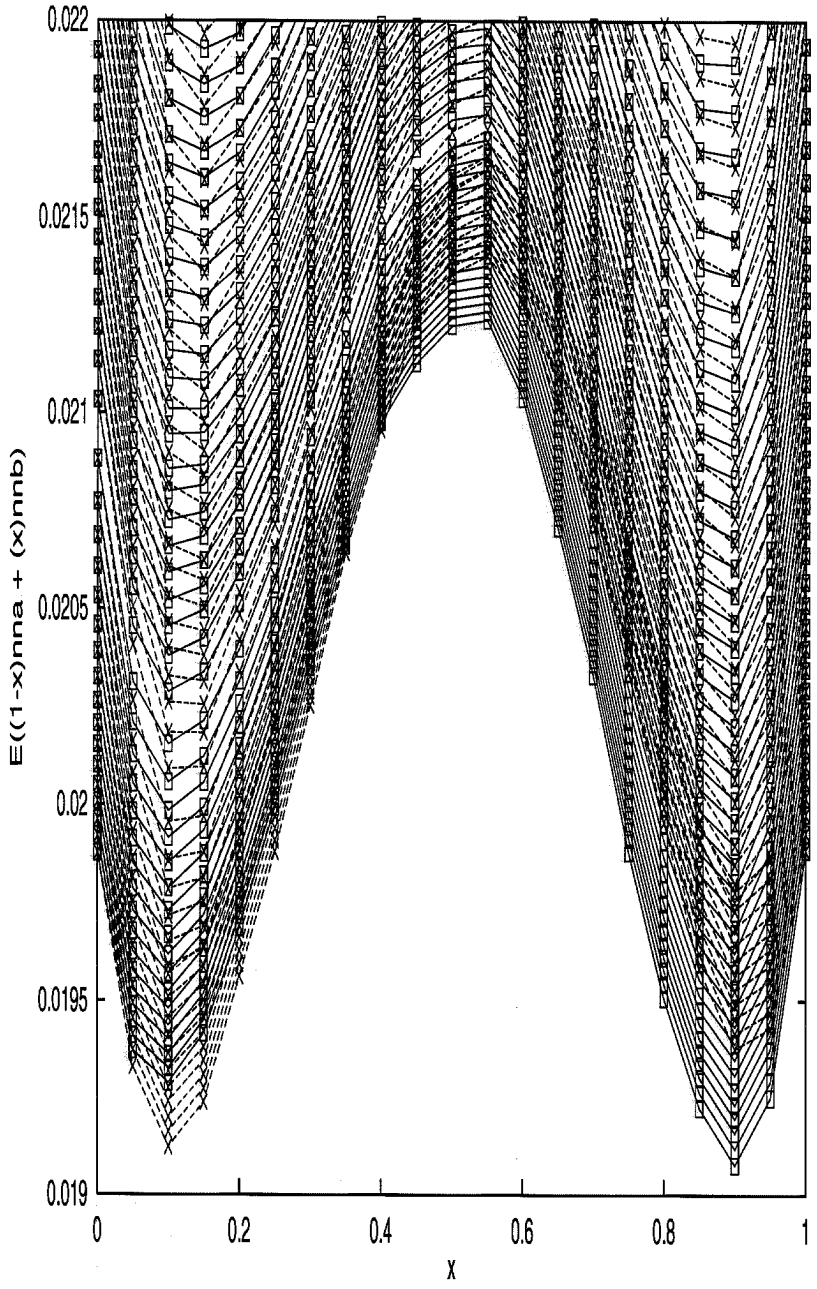

Figure 5.8: HUF and BNF Matching Performance After Many Epochs of Training, Average Over 50 Runs (Detail of Figure 5.7 – HUF error curves are labelled by squares. BNF error curves are labelled by x's.)

i. $nn^b := (1-\hat{\theta})n\hat{n}^a + (\hat{\theta})\hat{nn^b}$

e. Train

i. train nn^a on D_a for 20 epochs

ii. train nn^b on D_b for 20 epochs

Figure 5.9 shows the results of a single run, and Figure 5.10 is an enlargement of a band across the bottom of Figure 5.9, focusing on the later intervals of the run. Figures 5.13 and 5.14 show that the test errors on both students decreased monotonically during the run, so, in Figures 5.9 and 5.10, the endpoints representing student test error proceed down the sides of the figures as training progresses.

Note that in Figure 5.9, the lowest points on the initial matching curves are not as low as either endpoint of the next lower curve. Hence, more learning is accomplished by training the students for an interval on their separate data sets than by combining the students. This is no longer true later in the run – the error curves in Figure 5.10 dip far below the endpoints of the next lower curves.

In the combination process, the matched student networks are proportioned by a grid search so that the combined network minimizes error on the matching data set. Thus, the lowest point on the matching error curve indicates which convex combination is chosen to replace the students. The height of the test error curve at the lowest point on the matching error curve is the test error of the combination network. In general, the convex combination with the lowest error on the matching data set is not the convex combination with the lowest error on the test data set. However, the chosen combination network generally has a lower test error than either of the networks being combined. For example, in Figure 5.10, examine the bottom matching error and test error curves – output from the final combination. The lowest point on the matching error curve occurs at $\theta = 0.45$, so the grid search selects $(.55)n\hat{n}^a + (.45)\hat{nn^b}$ as the combination network. The test error of the combination network is the height of the test error curve at $\theta = 0.45$. This is lower than either endpoint of the test error curve, so the combination enhances learning.

In the later intervals, shown in Figure 5.10, each test error curve's endpoints are higher than the test error of the previous combination network, indicating that the interval of training on separate data sets increases the test errors for both copies of the combined network. Thus, each student overfits the data in its training set. But then each combination achieves a lower test error than the combination before.

Although the training phase increases test error, it is a necessary part of the process. If the training phase is not performed, then the test run consists of a series of combination phases. But in the combination phase, the combination network is duplicated to form the new students, so any convex combination of the untrained new students produces exactly the same network.

The training phase is necessary to derive the benefit of combination, but is

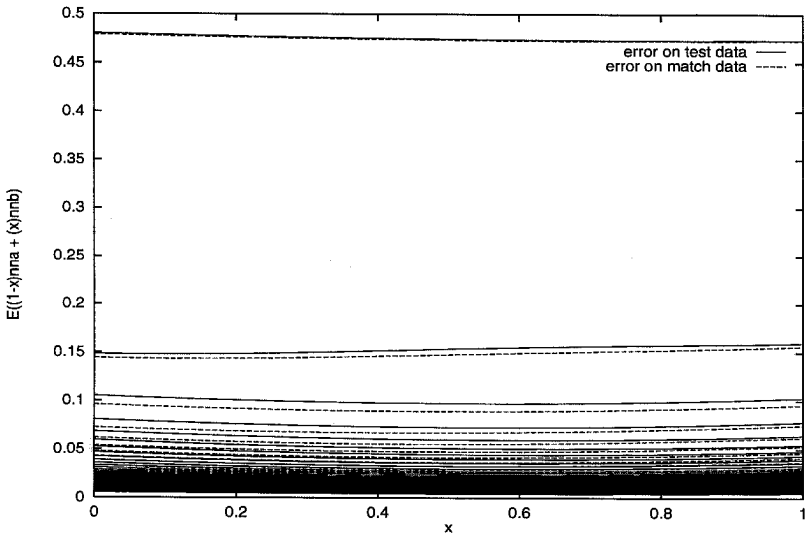

Figure 5.9: Matching at Work

the overfitting necessary? Perhaps the overfitting could be avoided or delayed by using the data examples differently. The combination data does not seem to play such an active role in training as the training data sets, and it does not seem to be overfit as early in training. Hence, learning may be enhanced either by shifting some examples from the matching data set to the training data sets or by sharing examples among the data sets, possibly reshuffling them among data sets between intervals.

Figures 5.11 and 5.12 show averages over 50 runs. They reflect the patterns found in Figures 5.9 and 5.10, indicating that the results for the single run are typical. In Figure 5.12, the alignment of the lowest point on each matching error curves with the lowest point on its corresponding test error curve is not typical of individual runs. It is an artifact of averaging – about half of the runs have the lowest matching error to the left of the lowest test error, and about half have the lowest matching error on the right.

Similar tests were run with BNF matching instead of HUF matching. The results are quite similar to those obtained with HUF matching.

5.3 Test Errors Over Intervals of the Test Runs

The error curves confirm that test errors decreased during training in all of the tests. Thus, the endpoints of the error curves proceed down the sides of the pre-

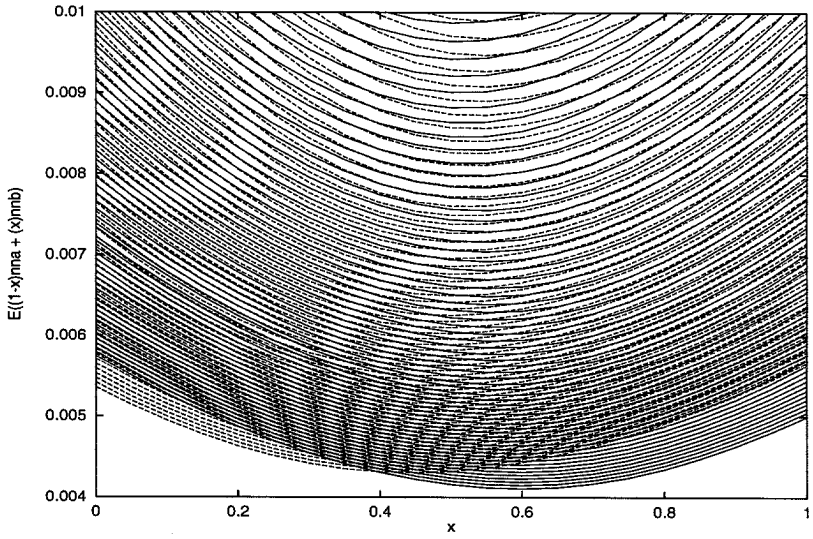

Figure 5.10: Matching at Work – Later Training Intervals. (Detail of Figure 5.9 – Test error is plotted with solid lines. Match error is plotted with broken lines.)

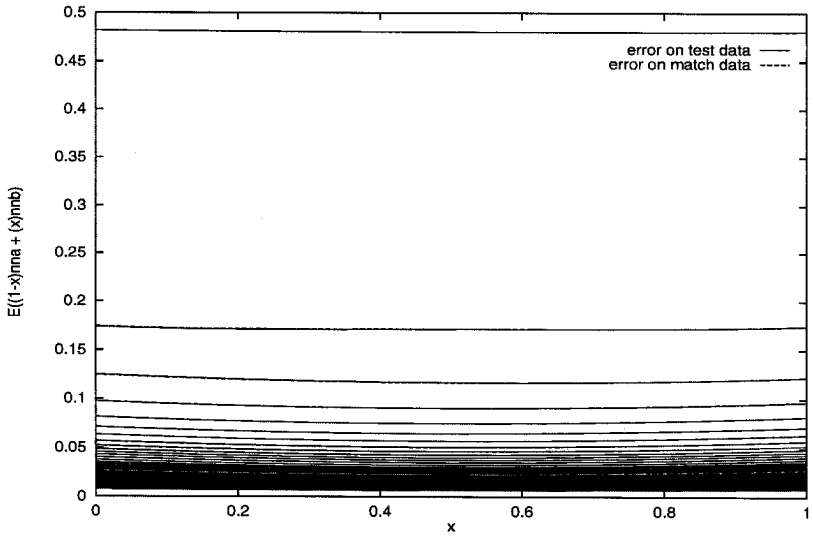

Figure 5.11: Matching at Work, Average Over 50 Runs

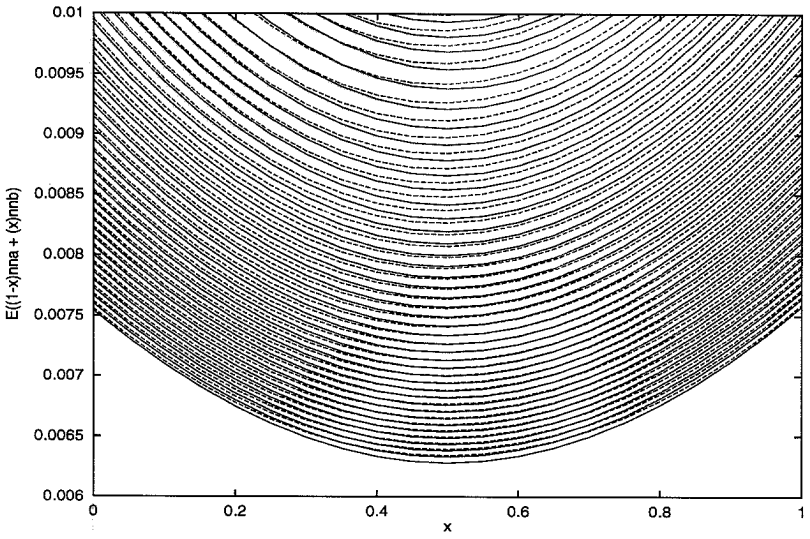

Figure 5.12: Matching at Work – Later Training Intervals, Average Over 50 Runs. (Detail of Figure 5.11 – Test error is plotted with solid lines. Match error is plotted with broken lines.)

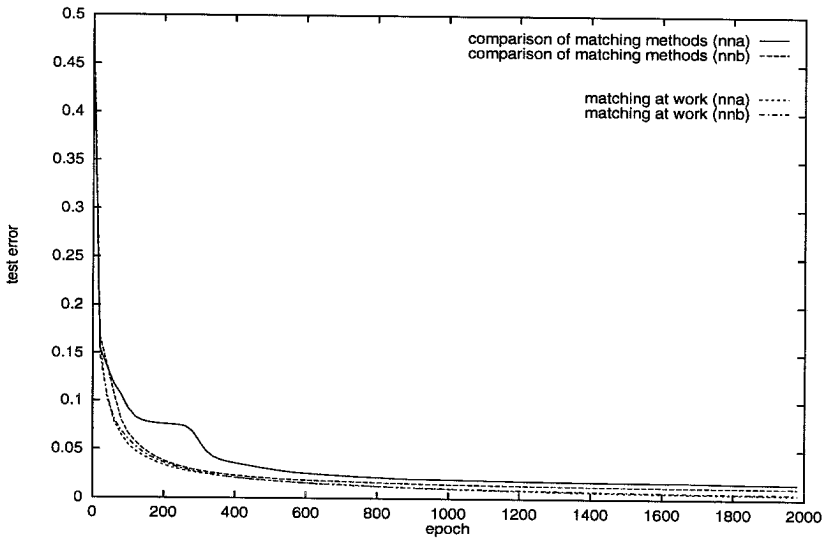

Figure 5.13: Network Test Errors as Training Progresses

vious figures as training progresses. In both the single run and the average over 50 runs, lower test errors were achieved by the test in which combination played a role in training (matching at work) than in the test in which combination networks were discarded (comparison of matching methods).

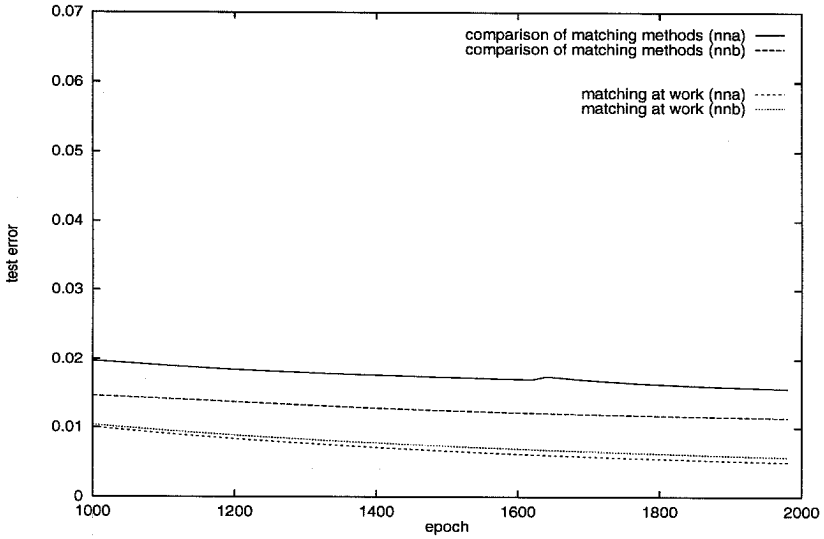

Figure 5.14: Test Errors as Training Progresses – Later Epochs (Detail of Figure 5.13)

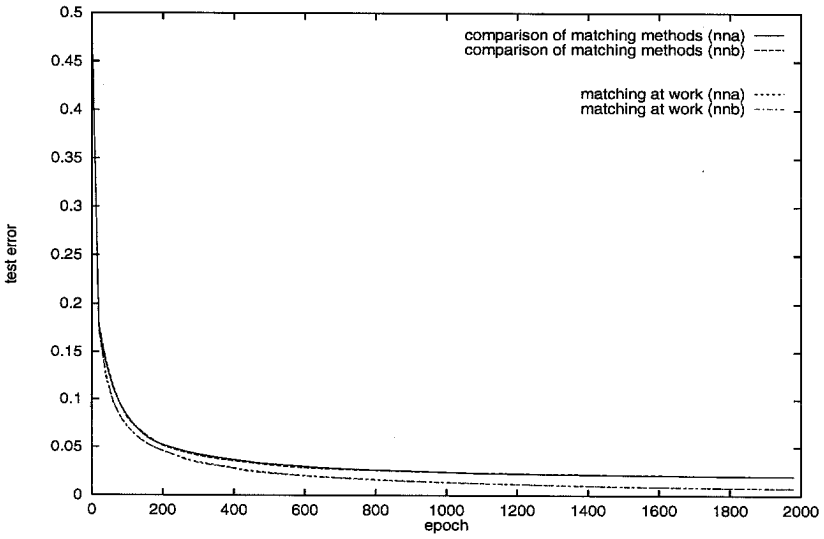

Figure 5.15: Network Test Errors as Training Progresses, Average Over 50 Runs

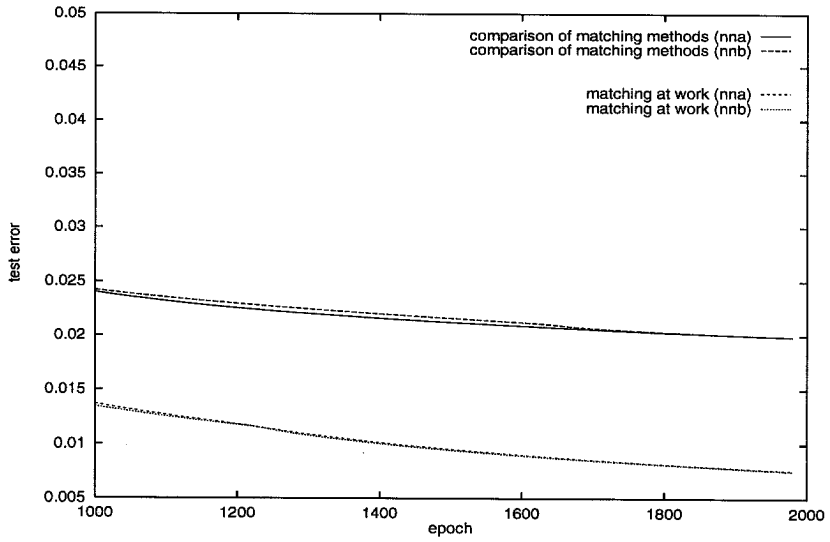

Figure 5.16: Test Errors as Training Progresses – Later Epochs, Average Over 50 Runs (Detail of Figure 5.15)

Chapter 6

Parameterized Networks

Network combination may have applications beyond training. The convex combination $(1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b$ of matched networks $n\hat{n}^a$ and $n\hat{n}^b$ may be viewed as a parameterized network in which θ can be adjusted as operating conditions change. For example, suppose two robots are to be trained to climb a hill. Each robot has a neural network which maps video camera input to motor power output. The robots only have enough memory to store a few data examples. The robots' networks train on the data examples experienced while the robots are piloted up the hill by human drivers. One robot trains on the sunny side of the hill, and the other one trains on the shady side. On the way up the hill, each robot records a few data examples. When the robots reach the summit, these examples are combined to form a data set, which is used to match the robots' networks. If nn^a is the network trained in sunny conditions and nn^b is the network trained in the shade, then the parameterized network can be tuned for specific conditions, increasing θ as lighting conditions brighten (see Figure 6.1). If the method is successful, then the convex combination networks will interpolate the behaviors of the original networks. Network extrapolation may be useful for training on nonstationary time series. A series of networks can be trained on progressive sliding windows of the data. After matching, the weights of the network series can be extrapolated to form a network for future action.

Figure 6.2 shows the results of a simple test of the effectiveness of matching to create parameterized networks. A 10-10-10 teacher network's weights were drawn uniformly at random from $[-10, 10]$. Then a perturbation network's weights were drawn at random from $[-1, 1]$. One network was trained on data generated by adding the perturbation network to the teacher network. Another network was trained on data generated by subtracting the perturbation network from the teacher network. The trained networks were matched by the HUF algorithm.

Different test data sets were generated by networks resulting from adding different multiples of the perturbation network to the teacher network. Then the

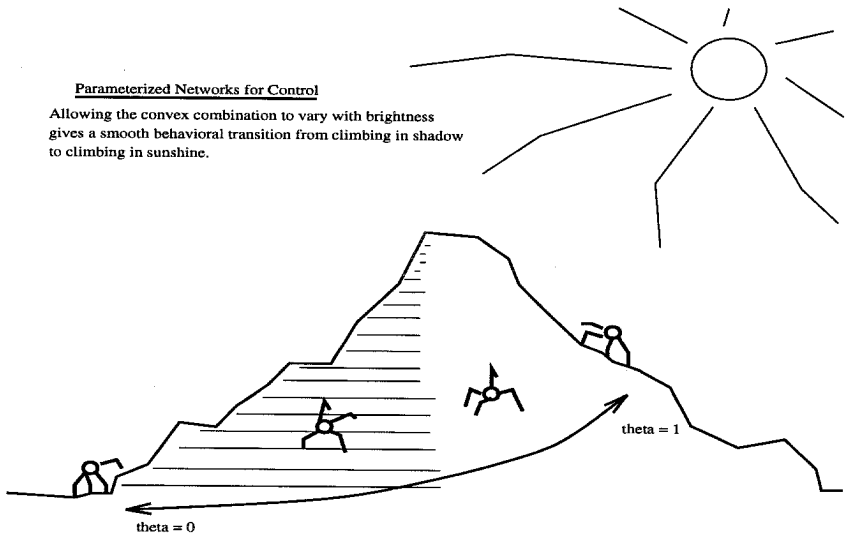

Figure 6.1: Parameterized Networks Can Be Tuned in Response to Changing Conditions

errors were evaluated for various linear combinations of the trained and matched networks. For each test data set, the error was minimized by the linear combination of trained networks corresponding to the degree of perturbation. For example, the test data generated by the teacher network with no perturbation was best fit by the equally weighted average of the trained networks. When a larger multiple than one of the perturbation network was added to the teacher network to generate the test data, the linear combination of trained networks formed by extrapolation performed best. Note that the lowest overall test error was achieved on data from the teacher network itself, which is the network exactly in the center of the networks that generated the training data.

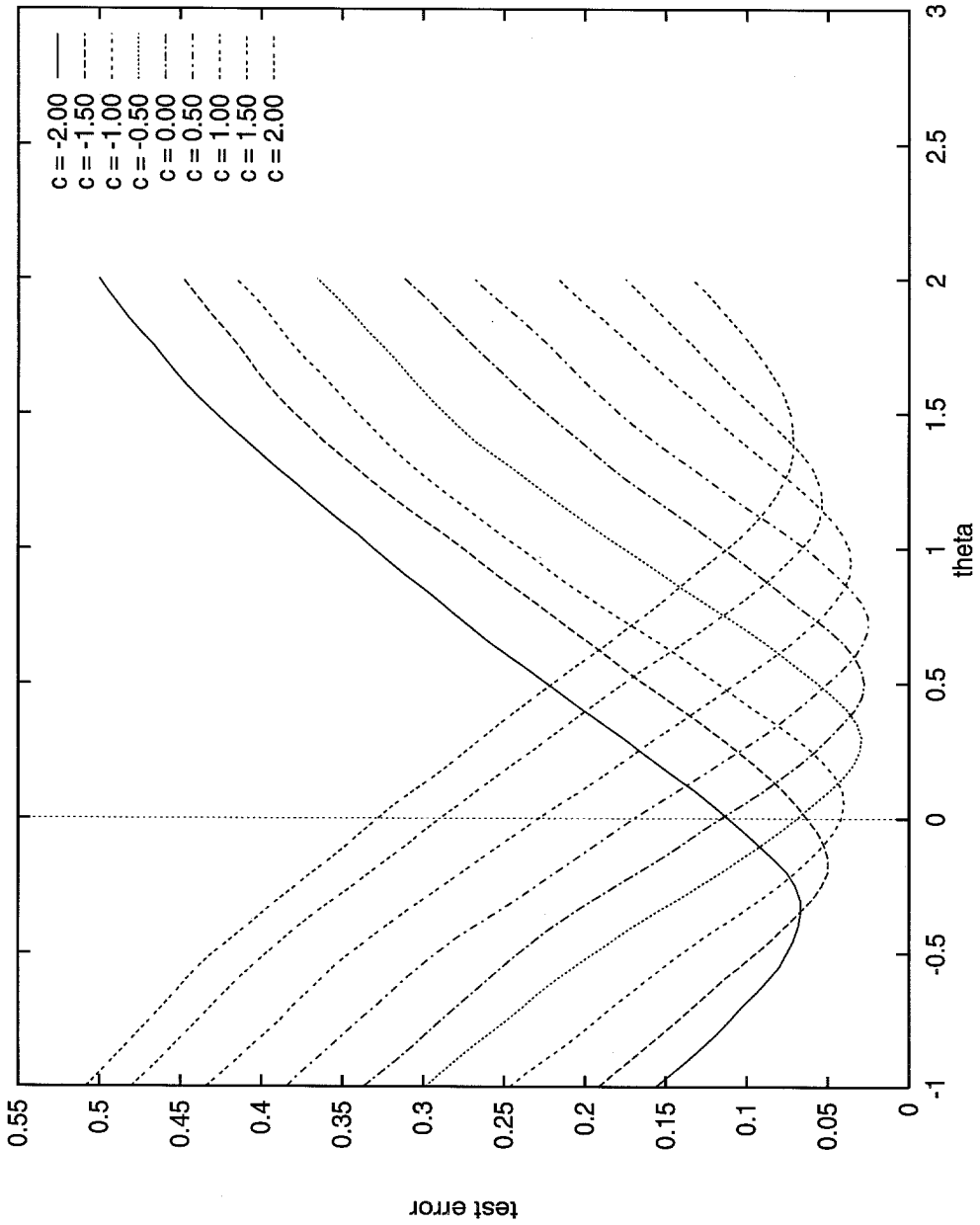

Figure 6.2: The best convex combination of matched networks tracks the perturbation of the teacher network.

Chapter 7

Feature Proportioning

A single-layer network with one output can be viewed as a feature recognizer. The network function is zero along a hyperplane in the input space. The function value increases to one as the input point recedes from the hyperplane on one side, and the function value decreases to negative one as the input point recedes on the other side. In a sense, the single unit reacts positively to inputs on one side of the hyperplane and negatively to points on the other side.

In a two-layer network, each hidden unit structure, consisting of a hidden unit's threshold and the weights to the hidden unit from the input units, functions as a single-layer network with one output. Thus, each hidden unit function can be viewed as a feature recognizer. Also, each output unit structure, consisiting of an output unit's threshold and the weights to the output unit from the hidden units, functions as a single-layer network. So each output unit structure can be viewed as a feature recognizer that operates on the space of hidden unit outputs. We will refer to these feature recognizers as features.

Suppose we train and match a pair of networks. Each network has learned some features well and some poorly. Suppose that one network has learned one half of the features well, and the other network has learned the other half of the features well. Then a convex combination of the networks will either do well on some features and poorly on others (if one network is heavily emphasized,) or the combination will have mediocre performance on all features (if the networks are equally emphasized.) It would be better to combine the features separately, in each case emphasizing the network that best learned the feature.

The following algorithm uses a pairwise network proportioning function to perform separate proportionings by features. Any of the proportioning algorithms discussed earlier can play the role of the function proportion(nn^a,nn^b). First, the trained and matched networks nn^a and nn^b are proportioned to form network nn. Then a network is formed by copying the weights of a single feature from nn^a to nn. Another network is formed by copying the weights of the corresponding feature from nn^b to nn. (The weights of feature (h,i) include the

threshold on the ith unit in layer h and the weights on all edges into the unit from the previous layer.) Proportioning these networks proportions only the single feature, since all other weights are equal. The result of this proportioning is assigned to nn. This process is performed for each feature.

```
feature_ proportion(nn^a, nn^b)
{
network nn, ~nn^a, ~nn^b
int h, i

nn := proportion(nn^a,nn^b)

for h := 1 to L
    for i := 1 to n_h
        ~nn^a := nn and ~nn^b := nn
        copy the weights of feature (h,i) from nn^a to ~nn^a
        copy the weights of feature (h,i) from nn^b to ~nn^b
        nn := proportion(~nn^a,~nn^b)

return nn
}
```

Figures 7.1 and 7.2 show the results of tests of the effectiveness of feature proportioning. In each test, a pair of networks were trained and then combined using feature proportioning. The test error was measured after the initial proportioning over all weights and after proportioning over each feature. Figure 7.1 shows the average and standard deviation of these test errors. The point plotted for feature 0 is the test error after proportioning by all weights. The other points show the test error after proportioning by successive individual features. The figure shows that the test error decreases during the course of feature proportioning. The reductions in test error due to proportionings are shown in Figure 7.2. The point plotted for feature zero is the difference between the minimum test error between the two trained networks and the test error of the network that results from the original proportioning involving all weights. The other points show the differences in test error after and before each feature is proportioned. On average, each proportioning improves the resulting network. The initial proportioning over all weights produces the greatest improvement. The first ten features are input-to-hidden unit features, and the remainder are hidden-to-output unit features. On average, the input-to-hidden unit proportionings reduce the test error more than the hidden-to-output unit proportionings.

The results are averaged over 100 tests. Each test used the twin networks framework. Each network had two layers of weights, with ten input units, ten

hidden units, and ten output units. Each data set input was drawn at random from $[-1, 1]$. Each test followed the procedure:

1. Create a teacher network by drawing each weight at random from $[-10, 10]$.
2. Create two student networks by drawing each of their weights at random from $[-0.1, 0.1]$.
3. Create a test data set of 2200 examples by feeding random input vectors to the teacher network to produce corresponding output vectors.
4. Create a training data set of 660 examples for each student network by feeding random input vectors to the teacher network to produce corresponding output vectors.
5. Train each student network on its data set. Train for 100 epochs, using sequential mode with random order of presentation of examples. Use backpropogation stepsize multiplier 0.01.
6. Combine the training data sets to form the data set to be used for matching and proportioning.
7. Match the trained networks by the HUF algorithm.
8. Proportion the trained networks, using hybrid proportioning with 5 grid partitions and 5 levels of binary search. Record the test error of the resulting network. Record the difference between this test error and the minimum test error of the two trained networks.
9. For each feature, proportion by the hybrid algorithm, with 5 partitions and 5 levels of binary search. After each feature, record the test error, and record the difference between this test error and the previous one.

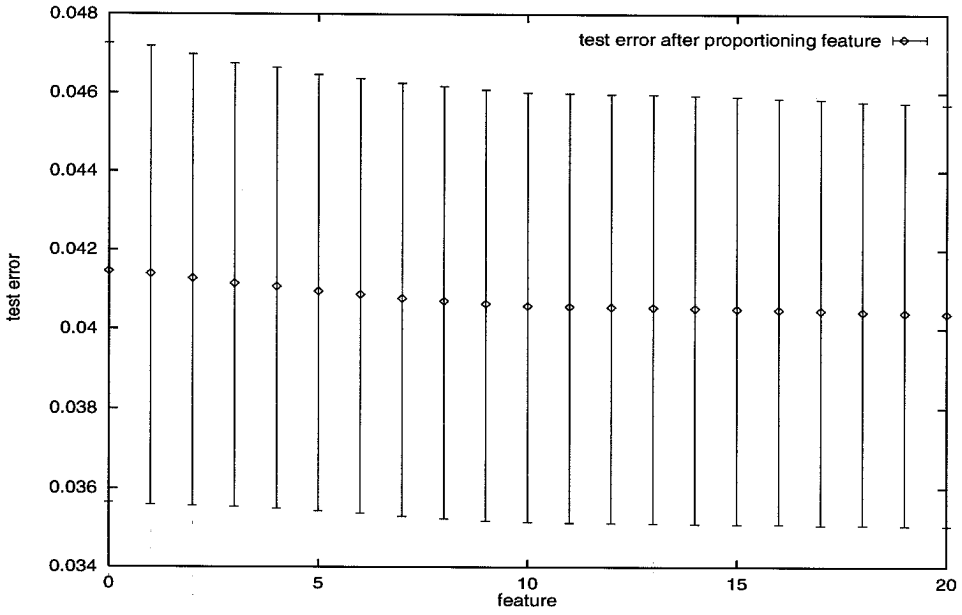

Figure 7.1: Test error decreases during the course of feature proportioning.

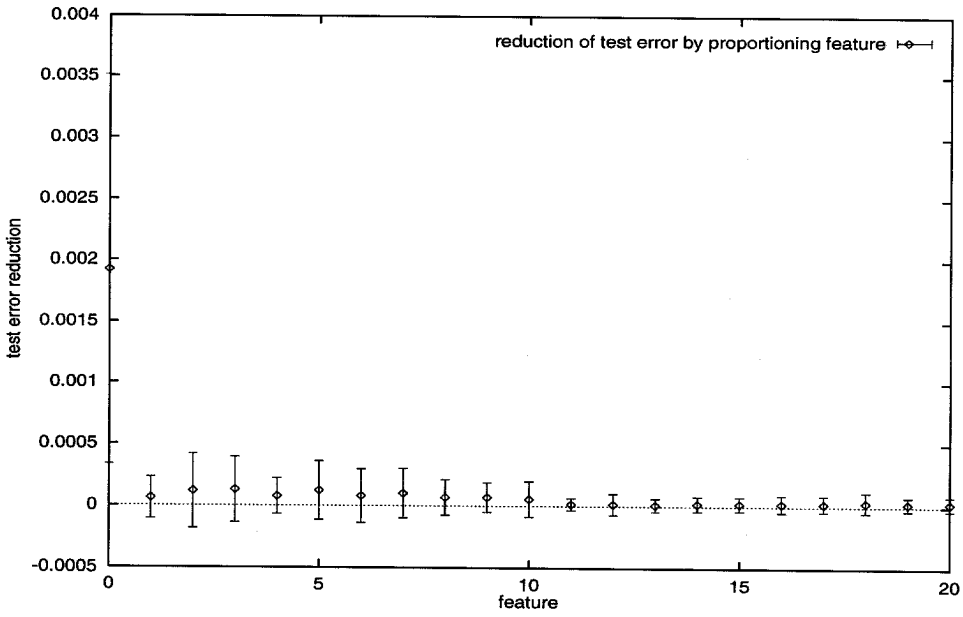

Figure 7.2: On average, each proportioning improves the network. The initial proportioning over all weights produces the greatest improvement. The input-to-hidden unit feature proportionings (the first ten features) produce more improvement than the hidden-to-output unit proportionings (the last ten features.)

Chapter 8

Multicomputer Implementation of Pairwise Combination

In this section we develop multicomputer implementations of matching and proportioning algorithms for pairs of networks. We assume that there is a multicomputer with a pair of processes, and the set of in-sample data examples to be used for training, matching, and proportioning are partitioned among the processes. The basic training process is shown in Figure 1.2. Each process has its own network. Each process trains its network for several epochs, sends a copy of its network to the other process, and receives a copy of the other process' network, leaving copies of both trained networks in both processes. Then the networks are matched and proportioned to form a combined network. The combined network replaces the processes' networks, and the training and combination process is repeated.

First, we present a model of multicomputer computation and its notation. Then we develop multicomputer implementations of HUF matching and grid and hybrid proportioning algorithms for pairs of networks. Next, we consider the cases in which the data is not strictly partitioned among the processes, i.e. the processes share some or all examples. Then we consider a relaxation of our implementation which sacrifices robustness for speed. Finally, we test some of the combination schemes considered in this section.

8.1 Multicomputer Model and Notation

We use the model of multicomputer computation presented by Van de Velde [18], with some changes in notation. (The model is only outlined here; for more

detail, see the books on concurrent computation by Van de Velde [18] and on UNITY by Chandy and Misra [3].) The model consists of sequential processes which communicate via channels. Each channel is unidirectional, i.e. it carries messages from a specific process to another specific process. Also, each channel preserves message order, i.e. messages sent on the same channel are received in the order in which they are sent. Between each pair of processes there is a pair of channels – one in each direction.

Each process has a unique process identifier. Each process stores as global variables its own identifier and the range of identifers of other processes. When discussing instances of variables that are duplicated across processes, we label instances with a superscript indicating their process. For example, suppose the processes have labels $\{0 \ldots 7\}$, and the variable nn is duplicated in each process. Then $\mathtt{nn}^5$ is the instance of variable nn in process 5.

Messages are passed along channels using the `send` and `receive` primitives. The `send` primitive has the syntax:

```
send M to p
```

where `M` is a variable or list of variables whose values form the message contents, and `p` is the identifier of the destination process. The `receive` primitive has the syntax:

```
receive M from p
```

where `M` is a variable or list of variables whose values are assigned by the message contents, and `p` is the identifier of the sending process. If the receiving variables are instantiated when the message is received, then the message contents replace the previous contents. If the receiving variables are not instantiated, then the receive primitive instantiates them and assigns them the values contained in the message.

In our notation, variables are not declared. Variables that are not received as parameters are assumed to be instantiated just prior to the first receipt of a value. Thus, it is illegal to access a variable before it is assigned.

To illustrate the notation, we present a function which sums the value of a duplicated variable over a pair of processes with identifiers 0 and 1.

```
p ∈ {0,1}
real sum_pair(real x; integer q;)
{
send x to q;
receive x' from q;

return x + x';
}
```

The first line contains the process identifier label p and the set of labels for all processes $\{0, 1\}$. Thus, there are two processes – process 0 and process 1, and each process refers to itself as process p and to the other process as process q.

The function definition uses syntax similar to that of the programming language C. The function type is declared, followed by the function name and the parameter list. All parameters are passed by value. Brackets surround the function body, and semicolons end statements.

In the function `sum_pair`, each process sends the value in its instance of x to the other process, and each process receives the value in the other process' instance by instantiating a variable x' and copying the message contents into it. Then the function returns the sum of the processes' instances.

The following call to `sum_pair` sums the values of the instances of variable z in processes 0 and 1, and it leaves the sum in each process' instance of variable y. Since the process identifiers are 0 and 1, each process refers to the other with process identifier $p \otimes 1$. (Throughout this paper, the symbol $\otimes$ refers to bitwise xor.)

$p \in \{0, 1\}$
$\vdots$
$y :=$ `sum_pair`$(z, p \otimes 1)$;
$\vdots$

The function `sum_pair` requires a single message exchange. On most message-passing multicomputers, the time required for a message exchange can be accurately modelled as the sum of an overhead time and a time that is proportional to the length of the message. Thus, it is more efficient to send a single message contining the values of several variables than it is to send the variables' values in several messages. So, when possible, we design our functions to compute the values of several variables, then transmit these values in a single message. The following function is an extension of `sum_pair` that sums sequences of variables that are duplicated in each process.

```
p ∈ {0,1}
real[] sum_sequence_pair(real x[0...n − 1]; integer q;)
{
send x[0...n − 1] to q;
receive x[0...n − 1]′ from q;

return (x[0] + x[0]′, ..., x[n − 1] + x[n − 1]′);
}
```

8.2 The Training Process – High Level Function

To supply a frame of reference for the development of matching and proportioning implementations, we present a high-level function to train a network on a pair of processes. In each process, the function is called with the following parameters.

- `nn` – the process' initial network
- `D` – the subset of the in-sample data stored in the process
- `intervals` – the number of training intervals to be performed
- `epochs` – the number of training epochs to be performed in each interval
- η – the backpropogation multiplier to be used in network updates $w := w - \eta\frac{\delta E}{\delta w}$ [11].

The value of `intervals` must be the same in both process' function calls. We assume that the values of `epochs` and η are also the same in both processes.

In each interval, each process trains its network for several epochs. The function `train`(nn, D, epochs, η) trains network nn on data set D for the specified number of epochs, using sequential mode backpropogation training with multiplier η. In each epoch, each example is presented to the network once, and the examples are presented to the network in random order.

After training, the processes distribute their networks, providing each process with instances of both networks. Next, the networks are swapped in processes 1 so that, in both processes, nn is the network trained in process 0, and nn′ is the network trained in process 1. This simplifies the rest of the function, allowing the matching and proportioning function calls to be the same in both processes, and allowing the variable θ to play the same role in both processes. Let nn^0 be the network trained in process 0, and let nn^1 be the network trained in process 1. Without the swap, $(1-\theta)\text{nn} + (\theta)\text{nn}'$ would mean $(1-\theta)nn^0 + (\theta)nn^1$ in process 0, and it would mean $(1-\theta)nn^1 + (\theta)nn^0$ in process 1.

To complete each interval, the process' networks are combined. The matching and proportioning functions that together perform combination are presented in the following sections.

```
p ∈ {0,1}
net train_intervals_pair(net nn; data D; integer intervals, epochs; real η;)
{
for i := 1 to intervals
    nn := train(nn, D, epochs, η);
```

```
    send nn to p ⊗ 1;
    receive nn' from p ⊗ 1;

    if p = 1 then swap(nn, nn');

    nn := match_pair(nn, nn', D, p ⊗ 1);
    θ := proportion_pair(nn, nn', D, p ⊗ 1);
    nn := (1 − θ)nn + (θ)nn';

return nn;
}
```

8.3 Implementation of Pairwise Matching

The following function implements the HUF matching algorithm that was developed earlier. In each process, all of the hidden unit pair similarities are computed for the data in the process. Then, in a single message exchange, all of the similarity measures are summed over the processes to find the similarities over all data.

The array a collects similarities. The variable a[h][i][j] stores the similarity between hidden unit i in layer h of the network trained in process 0 and hidden unit j in layer h of the network trained in process 1. Similarly, the array b collects similarities under sign flips. The similarity values are first computed over the data in each process, then summed over all data by summing over processes.

The function u[h][i]($\underline{x}$) returns the output of hidden unit i in layer h of network nn when the network is applied to input vector $\underline{x}$. Likewise, the function u[h][i]'($\underline{x}$) returns the output of hidden unit i in layer h of network nn'.

The function hungarian_method(S) returns the permutation matrix P that maximizes $\Sigma_{ij} s_{ij} p_{ij}$. The function permute_and_sign_flip_layer(h, P, F, nn) permutes the hidden units in layer h of network nn according to permutation matrix P and flips the signs of hidden units according to matrix F.

```
p ∈ {0, 1}
net match_pair(net nn, nn'; data D; integer q;)
{
for h := 1 to L − 1
    for i := 1 to n_h
        for j := 1 to n_h
            a[h][i][j] := −Σ_{x∈D}(u[h][i][0](x) − u[h][j][1]'(x))²;
            b[h][i][j] := −Σ_{x∈D}(u[h][i][0](x) + u[h][j][1]'(x))²;
```

```
(a[1...L-1][1...n_h][1...n_h], b[1...L-1][1...n_h][1...n_h]) :=
sum_sequence_pair(a[1...L-1][1...n_h][1...n_h], b[1...L-1][1...n_h][1...n_h], q);

for h := 1 to L-1
    for i := 1 to n_h
        for j := 1 to n_h
            s[i][j] := max(a[h][i][j], b[h][i][j]);
            if a[h][i][j] > b[h][i][j] then f[i][j] := 0; else f[i][j] := 1;
    P := hungarian_method(s[1...n_h][1...n_h]);
    nn := permute_and_sign_flip_layer(h, P, f[1...n_h][1...n_h], nn);

return nn;
}
```

8.4 Implementation of Pairwise Proportioning

Here we present multicomputer implementations of the grid and hybrid proportioning algorithms that were developed earlier. In each case, we develop support functions before implementing the proportioning method. First, we develop a support function to compute the error of a network that is duplicated among the processes on data that is distributed among the processes. Then we extend this function to operate on a sequence of networks. We use this function to implement grid proportioning. Similarly, we develop support functions to compute error gradients before implementing hybrid proportioning.

8.4.1 Grid Proportioning

Grid proportioning follows the procedure:

- Compute the error of various convex combinations of the matched networks.
- Return the convex combination with the lowest error.

When the proportioning function is called, the matched networks are duplicated in both processes. Hence, their convex combinations can be duplicated in each process. So we need a function to compute the error of a network that is duplicated among the processes on data that is partitioned among the processes. The following function performs this task, computing errors in each process, then summing over the processes. The sizes of the data sets in processes p and q, m_p and m_q, and the number of output units n_L are assumed to be global variables known in each process.

```
p ∈ {0, 1}
real E_pair(net nn; data D; integer q;)
{
E := Σ_{(x,y)∈D} ||nn(x) − y||²;
return 1/(2(m_p + m_q)n_L) sum_pair(E, q);
}
```

The following function is an extension of `E_pair` in which the errors are computed for a sequence of networks. To reduce communication, each process gathers the errors over its data for all networks; then these values are distributed among the processes in a single message exchange and summed.

```
p ∈ {0, 1}
real[] E_sequence_pair(net nn[0...n − 1]; data D; integer q;)
{
for i := 0 to n − 1 E[i] := Σ_{(x,y)∈D} ||nn[i](x) − y||²;
return 1/(2(m_p + m_q)n_L) sum_sequence_pair(E[0...n − 1], q);
}
```

The grid proportioning algorithm is implemented by the following function. The parameter $\triangle\theta$ is the grid spacing, i.e. the search space consists of $\theta \in \{0, \triangle\theta, 2\triangle\theta, \ldots, 1\}$. The function returns the value of θ that minimizes the error of convex combination network $(1-\theta)\mathtt{nn} + (\theta)\mathtt{nn}'$ over the in-sample data. Note that this function requires only a single message exchange, in `E_sequence_pair`.

```
p ∈ {0, 1}
real proportion_grid_pair(net nn, nn'; data D; real △θ; integer q;)
{
for k := 0 to 1/△θ nnθ[k] := (1 − θ)nn + (θ)nn';
E[0... 1/△θ] := E_sequence_pair(nnθ[0... 1/△θ], D, q);
return k△θ that minimizes E[k] over k ∈ {0... 1/△θ};
}
```

8.4.2 Hybrid Proportioning

Hybrid proportioning follows the procedure:

- Use a grid search to find domains of θ in which the error of convex combination network $(1-\theta)\mathtt{nn} + (\theta)\mathtt{nn}'$ is decreasing with respect to θ on the left boundary and increasing on the right boundary. (These domains contain local minima of error.)

- Use binary search to find a local error minimum in each domain.
- Return the value of θ corresponding to the least local error minimum.

To perform the grid search for domains with local minima, we develop functions to compute the gradient of the error with respect to the network weights ($\frac{\partial E}{\partial nn}$) and the gradient of the error with respect to theta ($\frac{\partial E}{\partial \theta}$).

The following function computes the error gradient with respect to the weights of network nn. Each process computes the gradient with respect its data; then the gradient is summed over the processes. The function backprop(nn, D) is an implementation of the backpropogation procedure [11]. The function sum_net_pair sums network weight vectors over processes. Its structure is the same as sum_pair, but it operates on networks instead of real numbers.

```
p ∈ {0, 1}
net grad_E_nn_pair(net nn; data D; integer q;)
{
grad := backprop(nn, D);
return sum_net_pair(grad, q);
}
```

```
p ∈ {0, 1}
net sum_net_pair(net nn; integer q;)
{
send nn to q;
receive nn′ from q;
return nn + nn′;
}
```

The following function computes the gradient of the error with respect to the network weights for several networks. To reduce communication, in-process computations are gathered and then distributed among the processes in a single message exchange. The function sum_net_sequence_pair sums a sequence of network weight vectors over the processes. It has the same structure as sum_sequence_pair, but it operates on networks instead of real numbers.

```
p ∈ {0, 1}
net[] grad_E_nn_sequence_pair(net nn[0 . . . n − 1]; data D; integer q;)
{
for i := 0 to n − 1 grad[i] := backprop(nn[i], D);
return sum_net_sequence_pair(grad[0 . . . n − 1], q);
}
```

```
p ∈ {0, 1}
net[] sum_net_sequence_pair(net nn[0...n − 1]; integer q;)
{
send nn[0...n − 1] to q;
receive nn[0...n − 1]′ from q;

return (nn[0] + nn[0]′, ..., nn[n − 1] + nn[n − 1]′);
}
```

Now we develop functions to compute the gradient of the error with respect to θ. Define $nn(\theta) \equiv (1-\theta)nn + (\theta)nn'$. Note that

$$\frac{\partial nn(\theta)}{\partial \theta} = \frac{\partial (1-\theta)nn + (\theta)nn'}{\partial \theta} = nn' - nn \tag{8.1}$$

Using the chain rule,

$$\frac{\partial E}{\partial \theta} = \frac{\partial E}{\partial nn(\theta)} \cdot \frac{\partial nn(\theta)}{\partial \theta} \tag{8.2}$$

$$= \frac{\partial E}{\partial nn(\theta)} \cdot (nn' - nn) \tag{8.3}$$

Hence, these functions are simple extensions of the previous functions to compute the gradient of the error with respect to network weights.

```
p ∈ {0, 1}
real grad_E_theta_pair(real θ, net nn, nn′; data D; integer q;)
{
return grad_E_nn_pair((1 − θ)nn + (θ)nn′, D, q) · (nn′ − nn);
}
```

```
p ∈ {0, 1}
real[] grad_E_theta_sequence_pair(real θ[0...n − 1]; net nn, nn′; data D; integer q;)
{
nnθ[0...n − 1] := ((1 − θ[0])nn + (θ[0])nn′, ..., (1 − θ[n − 1])nn + (θ[n − 1])nn′);
grad_E_nnθ[0...n − 1] := grad_E_nn_sequence_pair(nnθ[0...n − 1], D, q);
return (grad_E_nnθ[0] · (nn′ − nn), ..., grad_E_nnθ[n − 1] · (nn′ − nn));
}
```

The following function performs the binary search for a local minimum of the error of $\texttt{nn}(\theta)$ over $\theta \in (\texttt{left}, \texttt{right})$. Note that each recursive call requires a message exchange. Since the value of the error gradient in a function call determines the position at which the gradient will be evaluated in the next recursive

call, it is not possible to gather the in-process computations for all recursive calls and sum them all with a single message exchange. However, if message overhead costs are high enough, it is practical to compute and gather the in-process error gradients for all possible positions needed by a few levels of recursion and sum these values over processes with a single message exchange. Each process can store the sums and access only the values needed in the recursion.

```
p ∈ {0, 1}
real binary_pair(real left, right; integer levels; net nn, nn'; data D; integer q;)
{
center := (left+right)/2;
if levels = 0 return center;
if grad_E_theta(center, nn, nn', D, q) ≤ 0
    then return binary_pair(center, right, levels − 1, nn, nn', D, q);
    else return binary_pair(left, center, levels − 1, nn, nn', D, q);
}
```

The following function performs hybrid proportioning. The parameter $\triangle\theta$ is the size of each θ domain, and the parameter `levels` is the depth of binary search in each domain which is searched for a local error minimum. The variables θ`min`$[0 \ldots \frac{1}{\triangle\theta} - 1]$ store the local error minima positions, and the variables `E`$[0 \ldots \frac{1}{\triangle\theta} - 1]$ store the local error minima. This function requires several message exchanges. One message exchange is required to evaluate error gradients over the grid (to identify binary search domains). Each binary search requires one message exchange per level. So, if every grid space is searched, then the function performs $1 + \frac{1}{\triangle\theta}$`levels` message exchanges.

```
p ∈ {0, 1}
real proportion_hybrid_pair(real △θ; integer levels; net nn, nn'; data D; integer q;)
{
grad_E_theta[0 ... 1/△θ] := grad_E_theta_sequence_pair(θ[0 ... 1/△θ], nn, nn', D, q);
for k := 0 to 1/△θ − 1
    if grad_E_theta[k] ≤ 0 and grad_E_theta[k + 1] ≥ 0
        then θmin[k] := binary_pair(k△θ, (k + 1)△θ, levels, nn, nn', D, q);
        else θmin[k] := 0;

for k := 0 to 1/△θ − 1 nnmin[k] := (1 − θmin[k])nn + (θmin[k])nn';
E[0 ... 1/△θ − 1] := E_sequence_pair(nnmin[0 ... 1/△θ − 1], D, q);

return θmin[k] for k that minimizes E[k] over k ∈ {0 ... 1/△θ − 1};
}
```

8.5 Data Distribution Issues

Data need not be strictly partitioned among the processes. If there is a shortage of process memory, or, equivalently, an abundance of data, then not all of the data can be placed on the processes at once. If there is an abundance of process memory, then the processes can store overlapping subsets of the data.

In the case of scarce process memory, examples can be rotated between process memory and permanent storage. The optimal rate of rotation depends on machine-specific parameters, including the rate at which processes can read from permanent storage, and on problem-specific parameters, including the network architecture's tendency to overfit small data sets. Intuitively, the learning process works best when the greatest possible variety of in-sample data is in use. Hence, it is probably best to avoid giving processes overlapping data sets when there is not enough memory to place all in-sample data in process memory.

In the case of abundant process memory, the in-sample data can be partitioned among the processes, then each process can store some "extra" examples that are already found in another process. Using the extra examples will improve the network training within each process. However, the extra examples should not be used for error evaluations, matching, or proportioning. This is because error evaluation, matching, and proportioning are built on the assumption that the in-sample data set estimates the distribution of out-of-sample data. The extra data examples would be over-emphasized if they were used in these calculations.

In the extreme case of abundant process memory, all in-sample data can be duplicated on each process. In this case, error evaluations, matching, and proportioning can all be performed within each processor – there is no need for message exchanges to sum errors or error gradients over processes. Hence, the only communication required to perform an interval of training and combination is the single message exchange needed to distribute the trained networks among the processes.

8.6 In-Process Matching and Proportioning

Even when the in-sample data is not duplicated on each process, matching and proportioning can be performed without message exchanges by using only the data within each process to evaluate errors and error gradients. This approach trades robustness for speed – calculations based on the data will be less accurate since only in-process data is used, but only one message exchange occurs per interval, to distribute the trained networks. Since different in-process data leads to different outcomes of the matching and proportioning algorithms, this approach generally produces different combination networks in different processes. This is not a problem; the processes already develop different networks through in-process training, so there is no need for processes to begin each training inter-

val with identical networks. To produce a single trained network at the end of the training process, for the network combination to complete the final interval, revert to the more robust methods that sum errors and error gradients over all processes.

8.7 Tests

We present results that compare four of the training schemes discussed in this section:

1. Basic – The in-sample data is partitioned between processes, and the processes perform the matching and proportioning implementations presented in the text.

2. Overlapping Data – The in-sample data is partitioned between processes, then half of the data in each process is added to the other process. The extra data is used for training, but not for matching or proportioning. Only the partitioned data is used for matching and proportioning.

3. Duplicated Data – All of the in-sample data is duplicated in each process. So each process trains its network on all of the in-sample data. Matching and proportioning use all of the in-sample data, and these functions are performed without message exchanges. The only message exchange in each interval is to distribute the trained networks over the processes.

4. One Message – The in-sample data is partitioned between processes. In each interval, each process trains its network, the networks are distributed among processes by a message exchange, and then each process performs matching and proportioning using only its own portion of the data.

For each scheme, the results of a single training session are shown. Identical in-sample data sets, test data sets, and initial networks are used for all schemes. The twin networks framework [1] is employed, with a 10-10-10 teacher network with weights drawn uniformly at random from [-10,10]. The 10-10-10 student networks' initial weights are drawn uniformly at random from [-0.1,0.1]. The learning rate η is 0.01. There are 1320 examples in the in-sample data set (6 examples per weight) and 2200 examples in the test data set (10 examples per weight.) There are 20 training epochs per interval. Each scheme uses hybrid proportioning, with $\Delta\theta = 0.2$ (5 partitions) and 5 levels of binary search.

The plots show the error of the network in process 0 on various data sets for the different schemes. First, we compare the schemes' test errors over training epochs. Then, for each scheme, we examine the error on the training data sets in the processes, on all in-sample data, and on the test data.

8.7.1 Test Errors

Figures 8.1 and 8.2 compare test errors for the schemes. Figure 8.1 focuses on the early epochs of training, and Figure 8.2 focuses on the later epochs. Throughout the training sessions, the overlapping data scheme performs better than the basic scheme, and the duplicated data scheme performs best. The one message scheme fares slightly worse than the basic scheme.

The schemes that employ more resources per epoch perform better. The overlapping data scheme requires more process memory than the basic scheme, and the extra data in each process means more computation per training epoch. The duplicated data scheme can be seen as the extreme case of overlapping data, requiring each process to have enough memory to store all in-sample data. Since each process has twice as many examples as in the basic scheme, each training epoch requires twice as much computation. However, once the data is duplicated, no message exchanges are needed to match and proportion the trained networks. So, depending on the number of training epochs per interval and the communication speed, in some situations the duplicated data scheme will require less time per interval than the basic scheme.

The one message scheme is the slowest learner (per epoch), and it employs the least resources. Each process stores the same data as in the basic scheme, and the training epochs require the same amount of computation. However, the number of message exchanges is slashed by performing matching and proportioning using only the data in each process. Thus, the only information that each process receives about the data in the other process is through the network trained in the other process. Neither process receives any direct error or error gradient information computed over the in-sample data in the other process.

8.7.2 Training, In-Sample, and Test Errors

Figures 8.3, 8.4, 8.5, and 8.6 show the errors of the network trained in process 0 on the process 0 training data, the process 1 training data, all in-sample data, and the test data. Each figure corresponds to a training scheme, and all of the figures focus on epochs late in training.

Figure 8.3 shows the results for the basic scheme. The networks are combined after every 20 epochs of training. Combination reduces the test error as well as the error on all in-sample data and the error on the data in process 1. Combination increases the error on the data in process 0 since the network is trained on this data, but combination is based on all in-sample data. Through in-process training, the network's error on its own training data decreases, but the other errors increase. Nonetheless, successive combinations yield lower and lower errors on the in-sample and test data.

Figure 8.4 shows the results for the overlapping data scheme. The behavior of the errors is not as heavily punctuated by combination as in the basic scheme. Since each process has more data, and the processes share data, matching and

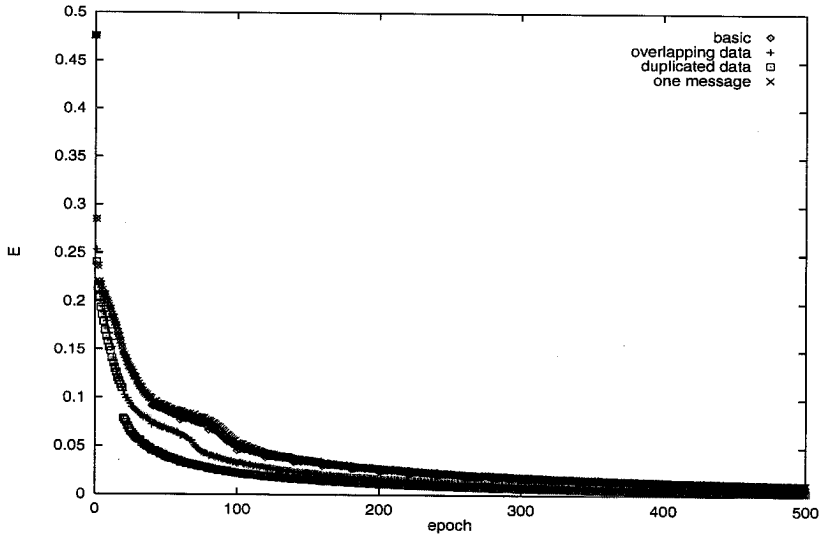

Figure 8.1: Test Errors for the Different Schemes – Early Epochs

proportioning on all data does not supply as much new information as it does in the basic scheme. On the other hand, the combination still noticeably decreases in-sample error, and it tends to decrease test error.

Figure 8.5 shows the results for the duplicated data scheme. In this scheme, the in-process data sets and the in-sample data set are all the same. Combination has an even more subtle effect than in the overlapping data scheme, but it still tends to decrease both in-sample and test error.

Figure 8.6 shows the results for the one message scheme. The error on the data in process 1 is much closer to the test error than in the other schemes, since the network trained in process 0 never receives direct exposure to error or error gradient evaluations over the data in process 1. Combination does not have much effect on the error over the training data in process 0. Combination does not always reduce the test error, but the reduction appears to be significant when it occurs.

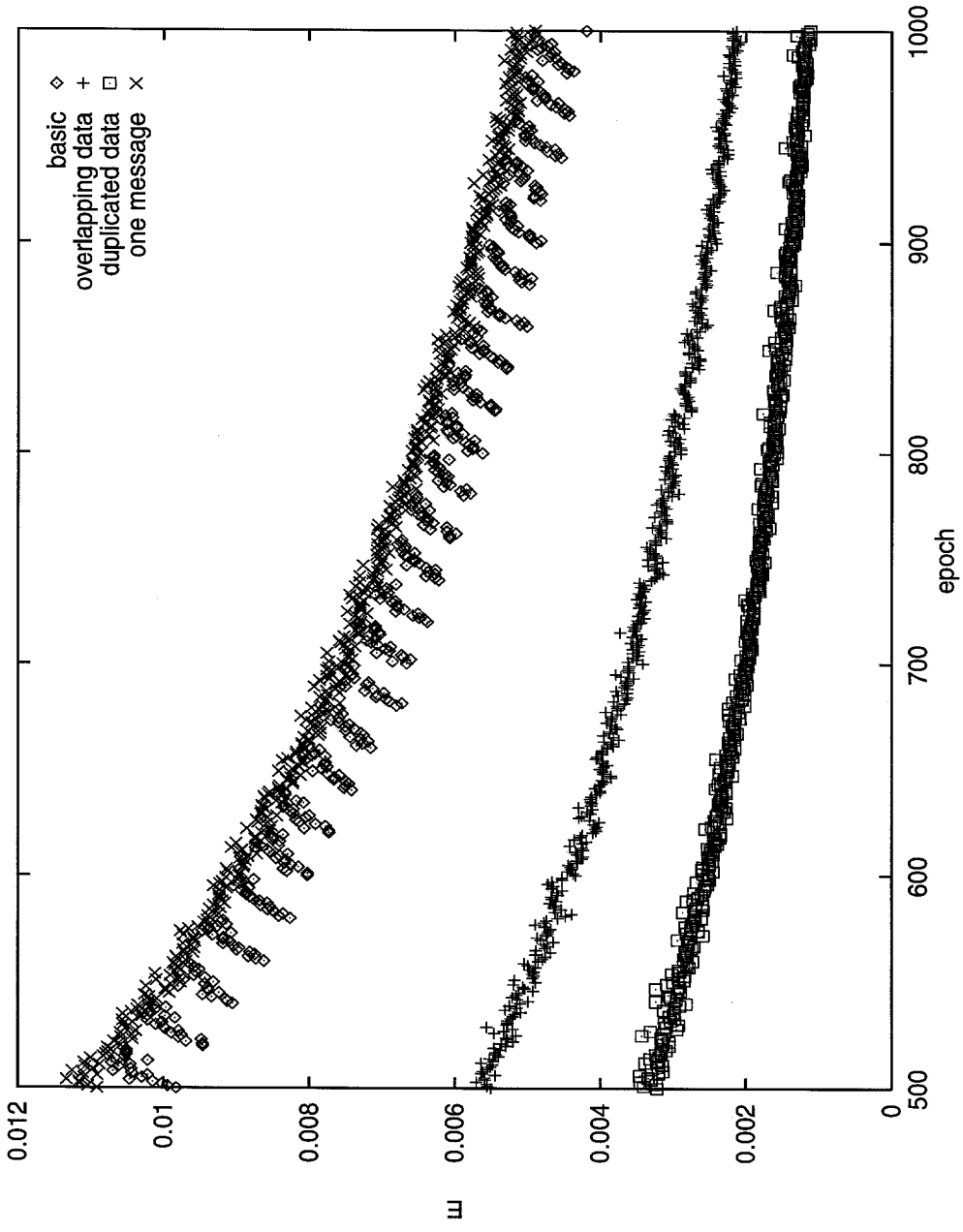

Figure 8.2: Test Errors for the Different Schemes – Late Epochs

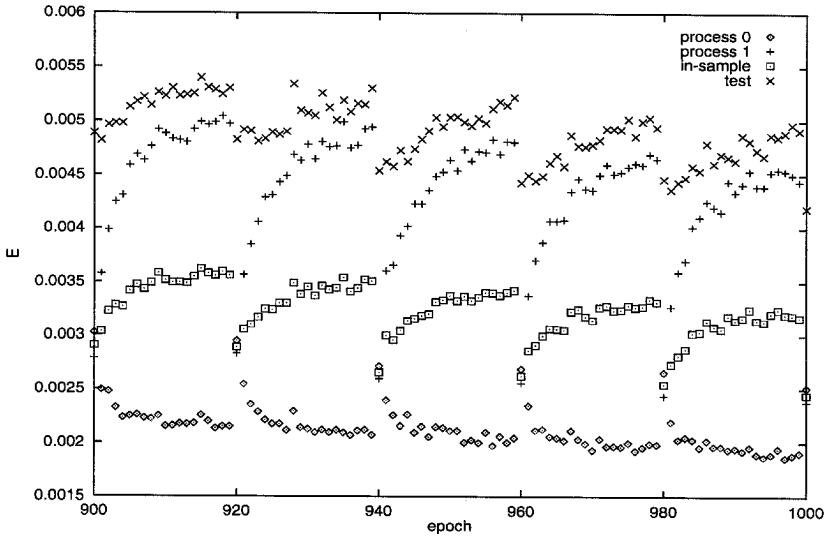

Figure 8.3: Basic Scheme – Errors in the Late Epochs of Training

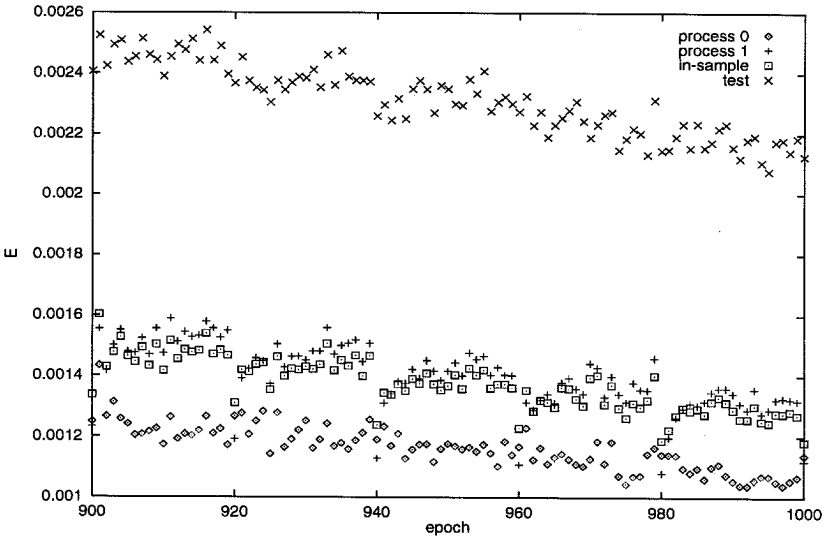

Figure 8.4: Overlapping Data Scheme – Errors in the Late Epochs of Training

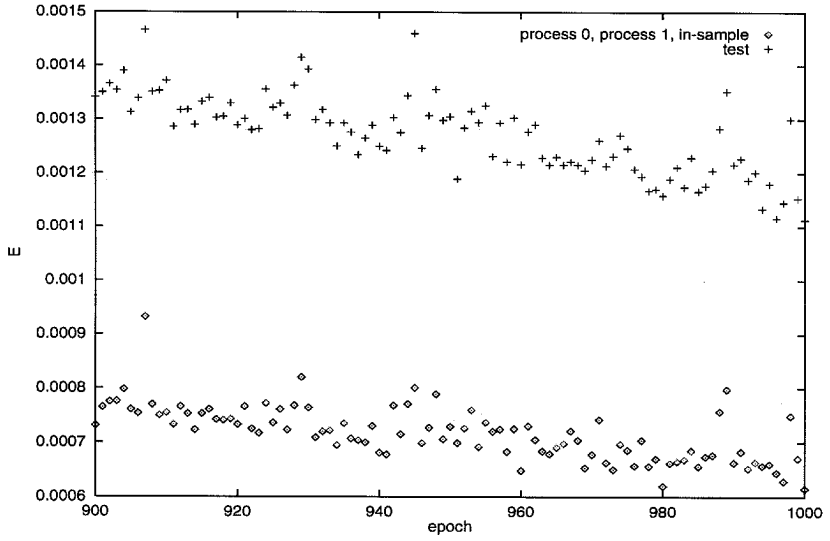

Figure 8.5: Duplicated Data Scheme – Errors in the Late Epochs of Training

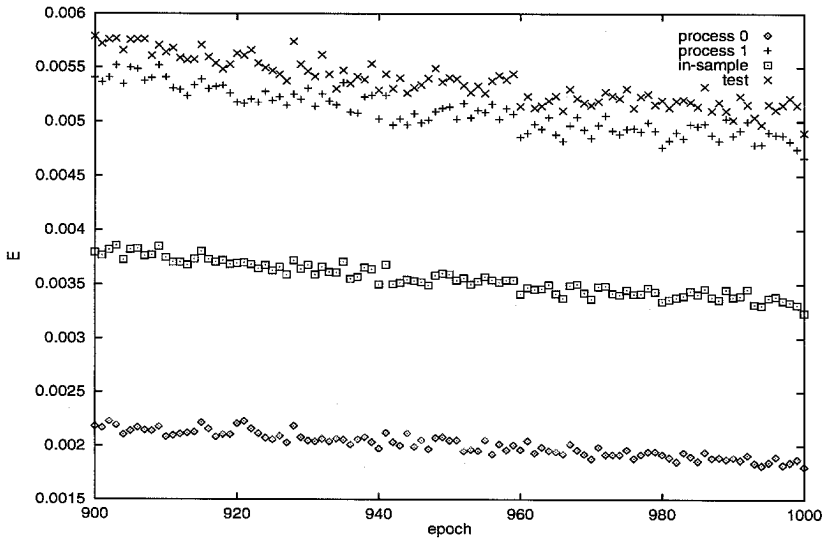

Figure 8.6: One Message Scheme – Errors in the Late Epochs of Training

8.8 Comparison of Multicomputer Training Schemes

The test in this section compares the two schemes for training a network on a multicomputer that were discussed in the introduction (see Figures 1.1 and 1.2.) In both schemes, the data is partitioned among processors.

In the gradient combination scheme:

- All processors have identical networks at all times.
- Each processor computes $\partial E/\partial w$ for its share of the data.
- After each training epoch, $\partial E/\partial w$ is summed over all processors, and the total $\partial E/\partial w$ is used to update the network in each processor.

The gradient combination scheme performs batch mode backpropogation.

In the network combination scheme:

- Processors' networks are initially identical.
- In each interval, each processor performs several epochs of backpropogation training on its network, causing processor networks to drift apart in weight space.
- After each training interval, the processor networks are combined to produce a network that replaces the network in each processor.

The test uses the twin networks framework [1], with 10-10-10 networks. Each teacher network weight is drawn independently and uniformly at random from $[-10, 10]$. Each initial student network weight is drawn independently and uniformly at random from $[-0.1, 0.1]$. All data set inputs are drawn uniformly at random from $[-1, 1]^{10}$. Each of the training sets D_a and D_b contains 660 examples, and the test data set contains 2200 examples.

The gradient combination test trains a single student on $D_a \cup D_b$. The backpropogation step size is 2.2, or .0033 per example. (This value gave the best training among 6.6, 4.4, 2.2, 1.1, and 0.5.)

The network combination test trains students separately on D_a and D_b for intervals of 100 epochs of sequential mode backpropogation with random order of presentation of examples. After each interval, the networks are matched by HUF, using $D_a \cup D_b$ as the matching data set. The matched networks are proportioned by a grid search over $\theta \in \{0.00, 0.05, \ldots, 1.00\}$, also using $D_a \cup D_b$ as the data set. The backpropogation step size is 0.01.

The results of a typical run are shown in Figures 8.7 and 8.8. The curve for the gradient combination scheme shows the student network's test error after each epoch of training. The curve for the network combination scheme shows the test error of nn^a (the network in processor a) after each epoch of training and after each combination. Examine Figure 8.8. The network combination scheme's error curve is disjoint, with each segment corresponding to a training

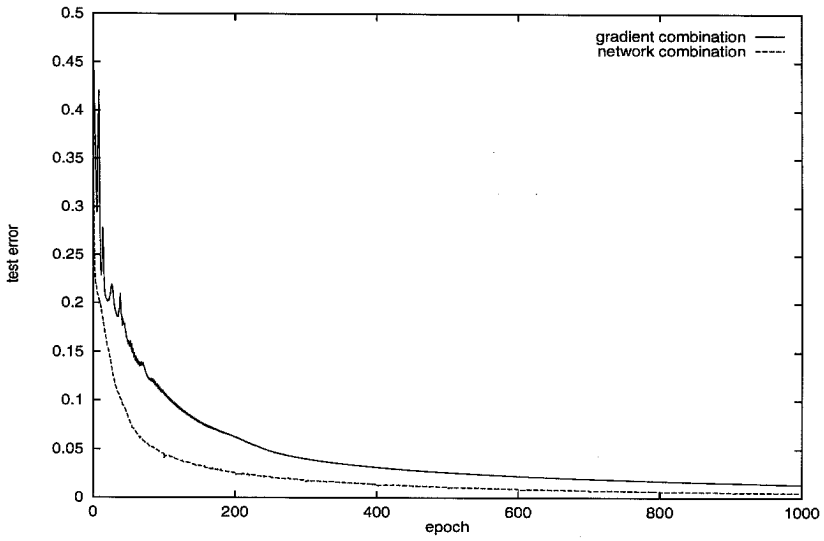

Figure 8.7: Comparison of Multicomputer Training Schemes

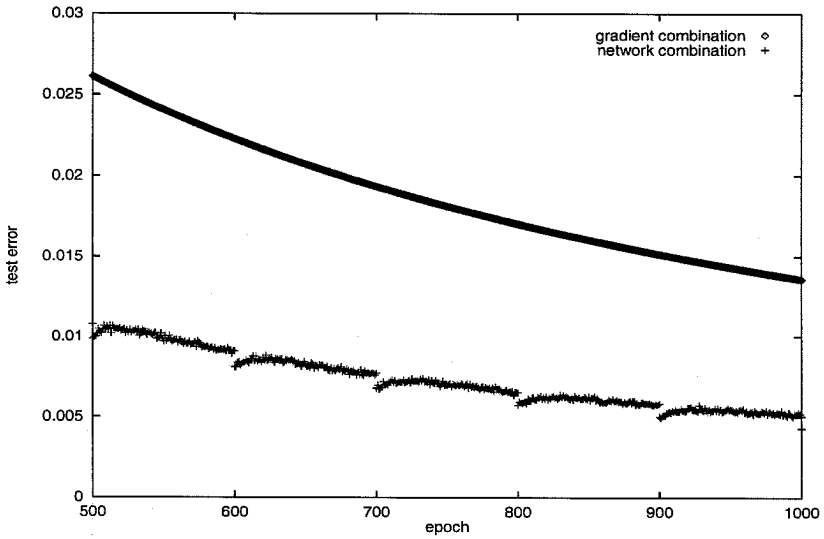

Figure 8.8: Comparison of Multicomputer Training Schemes – Later Epochs (Detail of Figure 8.7)

interval. Combination decreases test error. Then, the first few epochs of training on D_a increase the test error, but continued training eventually decreases the test error.

Epoch-by-epoch, network combination trains faster than gradient combination. This could be because the network combination scheme uses sequential mode training, while the gradient combination scheme must perform batch mode training. Comparison of the schemes' training speed is a machine-specific exercise which depends on the relative speeds of computation and communication. The gradient combination scheme requires less communication per training epoch, but more communication, than the network combination scheme. To evaluate the computation required by each scheme, define a "sweep" to be the computation required to apply a network to each example in training set D_a. Error evaluation requires one sweep. Backpropogation training requires a forward pass and a backward pass, so a training epoch requires two sweeps.

Examine the following algorithm for processor a to perform a training epoch in the gradient combination scheme:

gradient_combination_epoch_a(nn^a,D_a)
{
find $(\frac{\partial E}{\partial w})^a$ by backpropogation on nn^a and D_a
send $(\frac{\partial E}{\partial w})^a$ to processor b
receive $(\frac{\partial E}{\partial w})^b$ from processor b
$\frac{\partial E}{\partial w} := (\frac{\partial E}{\partial w})^a + (\frac{\partial E}{\partial w})^b$
$nn^a := nn^a - \eta \frac{\partial E}{\partial w}$
}

Each epoch requires 2 sweeps for backpropogation and 1 message exchange to combine gradients.

Now examine the following algorithm for processor a to perform one interval of training in the network combination scheme:

network_combination_interval_a(nn^a,D_a)
{
train nn^a on D_a for 100 epochs

send nn^a to processor b
receive nn^b from processor b

For each pair of hidden units with one unit in each network and both units in the same layer, compute **similarity**^{a} and **flipped_similarity**^{a} over D_a

send **similarity**^{a}'s and **flipped_similarity**^{a}'s to processor b
receive **similarity**^{b}'s and **flipped_similarity**^{b}'s from processor b

sum **similarity**^{a}'s and **similarity**^{b}'s to produce **similarity**'s
sum **flipped_similarity**^{a}'s and **flipped_similarity**^{b}'s to produce **flipped_similarity**'s
use **similarity**'s and **flipped_similarity**'s to match the networks,
producing $n\hat{n}^a$ and $n\hat{n}^b$

$\forall\theta \in \{0.00, 0.05, \ldots, 1.00\}$ find $E(\theta)^a$, the error of $nn(\theta) = (1-\theta)n\hat{n}^a + (\theta)n\hat{n}^b$ on D_a

send the errors $E(0.00)^a, \ldots, E(1.00)^a$ to processor b
receive the errors $E(0.00)^b, \ldots, E(1.00)^b$ from processor b

$\forall\theta \in \{0.00, 0.05, \ldots, 1.00\}$ $E(\theta) := E(\theta)^a + E(\theta)^b$
$\hat{\theta} := \theta$ that minimizes $E(\theta)$
$nn^a := (1-\hat{\theta})n\hat{n}^a + (\hat{\theta})n\hat{n}^b$
}

Each interval requires 200 sweeps for training, 2 sweeps to compute the **similarity**'s and **flipped_similarity**'s, less than 1 sweep for matching, and 20 sweeps to compute the errors for proportioning. This makes 223 sweeps for 100 epochs of training, or 2.23 sweeps per epoch. Also, the interval requires 3 message exchanges, making .03 communications per training epoch.

For a given machine, let s be the computation time required for a sweep, and let c be the communication time required for a message exchange. To compare the error curves over time, the gradient combination curves in Figures 8.7 and 8.8 should be stretched along the x-axis by $2s + c$, and the network combination curves should be stretched by $2.23s + .03c$.

The algorithms and the test presented in this section admit many variations; they are not intended as a complete discourse on the implementation and evaluation of the multicomputer training schemes. The test and the analysis are intended only as a concrete example to convey some of the issues and tradeoffs in multicomputer training. Comparing the two schemes, we find that the main advantages of network combination are that it allows sequential mode training and that it requires substantially less communication per epoch of training. On the other hand, because of matching and proportioning, the network combination scheme requires a bit more computation per training epoch than the gradient combination scheme.

Chapter 9

Algorithms to Combine Several Networks

In the next few chapters, we explore methods to combine several networks. Our notation, algorithms, and implementations build on those used to combine pairs of networks. In this chapter, we develop algorithms. We begin with simple algorithms that combine several networks by iteratively combining pairs of networks. We proceed to more complex and robust algorithms. In the next few chapters, we develop multicomputer implementations of combination-based training, and we present performance results for these implementations.

9.1 Definitions and Notation

Let n be the number of networks to be combined. The networks are denoted $nn^0, \ldots, nn^{n-1}$. To streamline notation, some of the algorithms presented here implicitly assume that n is a power of 2. In every case, it is easy to adjust the algorithm for cases in which n is not a power of 2, and this adjustment will be made later when we present implementations of the algorithms. The set of data examples to be used for matching is denoted D_m, and the set of data examples to be used for proportioning is denoted D_p.

9.2 Iterated Pairwise Combination – Fan-In Combination

We can combine several networks by using any of the pairwise combination methods from the previous chapters as a basic operation. A simple way to do this is to accumulate combinations one at a time in the following manner.

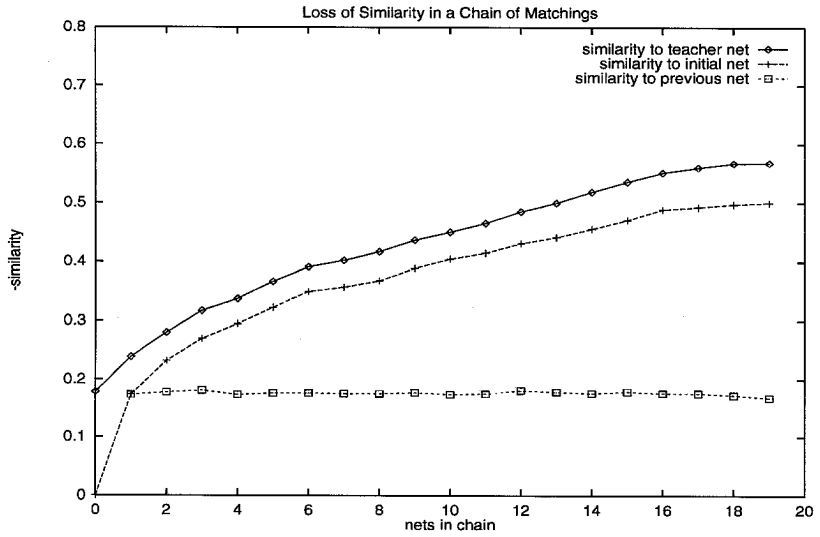

Figure 9.1: Similarity decreases over the course of a chain of matchings.

- $nn^{0,1}$:= combine(nn^0,nn^1)
- $nn^{0,1,2}$:= combine($nn^{0,1}$,nn^2)
- $nn^{0,1,2,3}$:= combine($nn^{0,1,2}$,nn^3)

⋮

- $nn^{0,\ldots,n-1}$:= combine($nn^{0,\ldots,n-2}$,nn^{n-1})

One problem with this method is that it emphasizes the later networks over the earlier ones. In the final combination network $nn^{0,\ldots,n-1}$, network nn^{n-1} has as much influence as all of the other networks combined.

The other problem with this method is that the quality of matchings degrades along a chain of matchings (see Figure 9.1.) This is something like the whisper game, in which one person tells a second person a secret, then the second person tells a third person the secret, etc. With every retelling, there are slight omissions and embellishments. By the time the tenth person hears the story, it bears no resemblance to the original tale.

We get a more balanced combination, and shorter chains of matchings, by pairing off networks, combining each pair, then repeating the process on the combined networks recursively (see Figure 9.2). This is called fan-in combination.

- $nn^{0,1} := \text{combine}(nn^0, nn^1)$, ..., $nn^{n-2,n-1} := \text{combine}(nn^{n-2}, nn^{n-1})$
- $nn^{0,3} := \text{combine}(nn^{0,1}, nn^{2,3})$, ..., $nn^{n-4,n-1} := \text{combine}(nn^{n-4,n-3}, nn^{n-2,n-1})$

⋮

-
- $nn^{0,\ldots,n-1} := \text{combine}(nn^{0,\frac{n}{2}-1}, nn^{\frac{n}{2},n-1})$

Both fan-in combination and combination by accumulation require $n-1$ pairwise combinations. Fan-in combination has the advantage that some of its pairwise combinations can be performed simultaneously, while combination by accumulation must perform its pairwise combinations sequentially.

9.3 Independent Matching and Proportioning

In this section, we explore methods to combine several networks by first matching all of the networks, then proportioning the matched networks. Matching several networks allows us to check the consistency of matchings between pairs of networks. Proportioning n networks is a a nonlinear optimization problem in a space with $n-1$ dimensions. Fan-in combination performs a degenerate search in this space, exploring only one dimension at a time. We will develop more complex and more robust algorithms that search higher-dimensional neighborhoods for the best convex combination.

9.4 Matching Several Networks

9.4.1 Iterated Pairwise Matching

Matching can be performed by iterating a pairwise matching algorithm. Define match(nn^a,nn^b) to be a pairwise matching procedure that rearranges network nn^b.

Pass-It-On Matching

We could match the networks in succession, rearranging the second network to match the first, then rearranging the third to match the second, etc. We call this the pass-it-on algorithm.

- match(nn^0,nn^1)
- match(nn^1,nn^2)

⋮

-

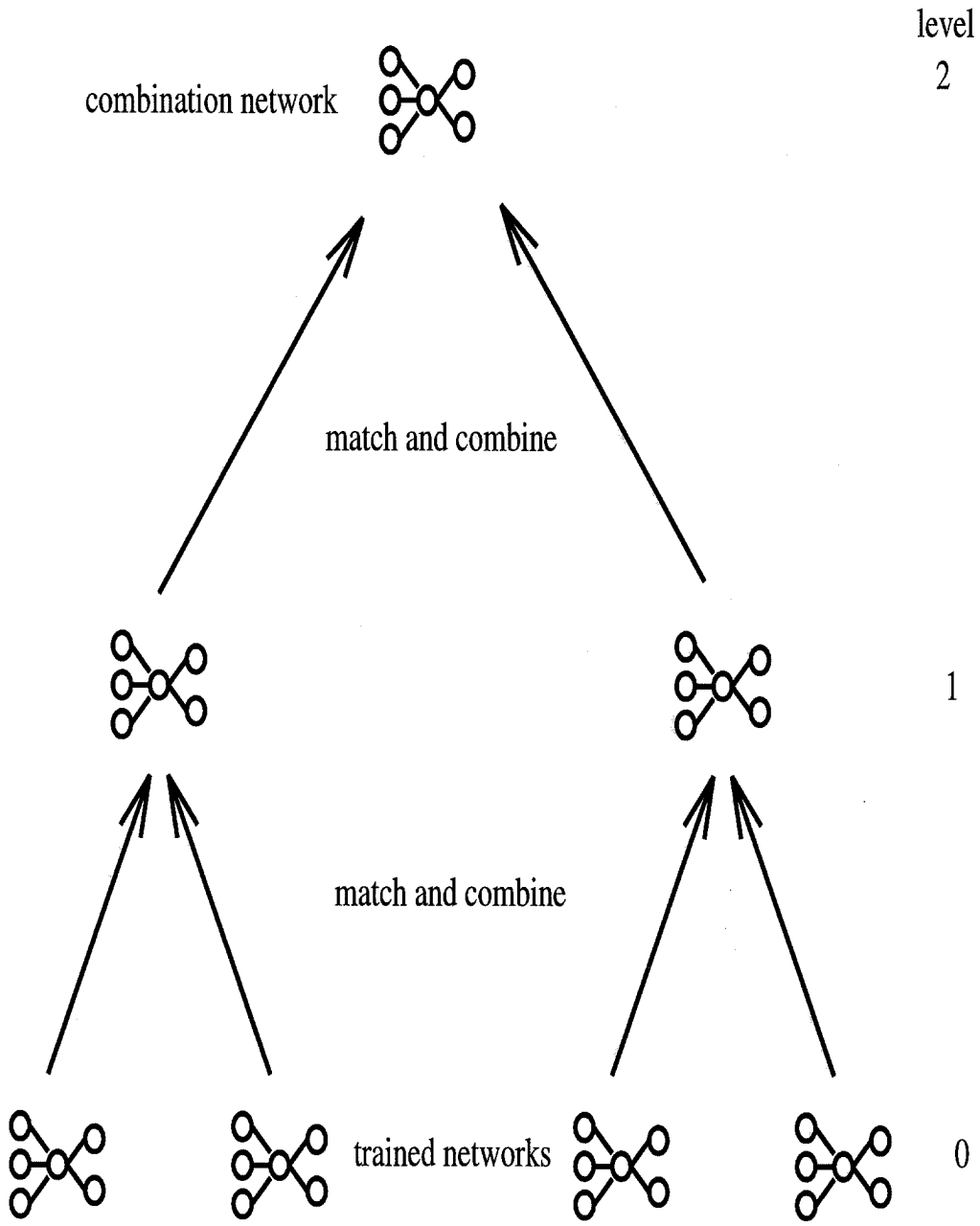

Figure 9.2: Fan-in combination.

- match(nn^{n-2},nn^{n-1})

This method has a long chain of matchings, which encourages loss of matching quality (see Figure 9.1.) There are $n-1$ matchings from nn^0 to nn^{n-1}, so these networks are not likely to be well matched by the procedure.

Busybody Matching

To reduce the number of matchings separating networks, we can match each network to one of the networks, for example nn^0. We call this the busybody algorithm.

- match(nn^0,nn^1)
- match(nn^0,nn^2)

⋮

-
- match(nn^0,nn^{n-1})

A single matching separates each network from nn^0, so the maximum number of matchings separating any pair of networks is two. This method emphasizes nn^0 much more than the other networks.

Fan-Out Matching

Another option is to match one network to another, then match networks to each of the two matched networks, then match networks to each of the four matched networks, etc., spreading matching like a communicable disease. We refer to this as fan-out matching, since the matching fans out from nn^0 in a binary tree.

- match(nn^0,$nn^{\frac{n}{2}}$)
- match(nn^0,$nn^{\frac{n}{4}}$), match($nn^{\frac{n}{2}}$,$nn^{\frac{3n}{4}}$)

⋮

-
- match(nn^0,nn^1), ..., match(nn^{n-2},nn^{n-1})

No more than $\log n$ matchings separate any network from nn^0, so no more than $2\log n$ matchings separate any pair of networks.

9.4.2 Checking for Consistency

If we have a way of measuring the functional similarity of corresponding weights in a pair networks, then we can use the measure to evaluate whether or not a pair of networks are well-matched. The following similarity metric compares the activities of corresponding hidden units over a set of data; it is the same metric that is maximized in the HUF algorithm. The similarity of networks nn^a and nn^b on data set D is defined:

$$S(nn^a, nn^b) \equiv -\Sigma_{\underline{x} \in D} \Sigma_{h \in \{1 \ldots L-1\}} \Sigma_{i \in \{1 \ldots n_h\}} (u^a_{hi}(\underline{x}) - u^b_{hi}(\underline{x}))^2 \qquad (9.1)$$

where $u^a_{hi}(\underline{x})$ and $u^b_{hi}(\underline{x})$ are the outputs of the ith hidden units in layer h of networks nn^a and nn^b, respectively.

This metric allows us to evaluate the loss of fidelity along a chain of matchings (see Figure 9.1.) Thus, it allows us to evaluate matchings of several networks. Since the matching algorithms we have presented leave much freedom for reordering their chains of matchings, we can try an algorithm, then test its effectiveness by measuring similarities among pairs of networks. If the networks are not well-matched, then we can permute the roles played by various networks in the matching algorithm, and perform the matching algorithm once again.

The consistency checks can be incremental, with failure followed by local permutations of the chain of matchings. In the pass-it-on algorithm, after each matching, we can evaluate the similarity between the newly matched network and the second network back in the chain. If these networks are not well-matched, then we can remove the newly matched network from the chain and try some other network that is not in the chain. If all candidates fail to match well, then the last network in the chain should be removed, and the process should be repeated.

Also, the consistency checks can follow the complete algorithm, with failure followed by global permutations of the chain of matchings. After the busybody algorithm, the average similarity can be evaluated over randomly selected pairs of networks. If the networks are badly matched, then the algorithm can be performed again with a different "busybody" network. A similar measurement can be applied after the fan-out algorithm. If the networks are badly matched, then the network labels $0 \ldots n-1$ can be permuted randomly, and the algorithm can be performed again, yielding a different order and hierarchy of matchings.

9.4.3 Matching by Average Similarity

The pairwise matching algorithms can be altered to match a single network to a group of previously matched networks, instead of matching to a single network. For each layer, calculate the similarity matrices between the single network and each of the matched networks. Then use the element-by-element average of the similarity matrices to identify the permutations and sign flips that best align the single network's weights with the matched networks' weights.

Let nn be the network to be matched to the set of networks $\{nn^0, \ldots, nn^{m-1}\}$. Let u_{hi} denote the function of hidden unit i in layer h of nn, and let u^k_{hj} denote the function of hidden unit j in layer h of network nn^k. Let D be the data set used for matching. The HUF algorithm can be extended as follows.

HUF_set(nn, $\{nn^0, \ldots, nn^{m-1}\}$, D)
{
matrix S, F, P
real **similarity**, **flipped_similarity**
int h, i, j

$\forall\ h \in \{1 \ldots L-1\}$
  $\forall\ (i,j) \in \{1 \ldots n_h\} \times \{1 \ldots n_h\}$
    similarity $:= -\frac{1}{m}\Sigma_{k=0}^{m-1}[\Sigma_{\underline{x} \in D}\ (u_{hi}(\underline{x}) - u^k_{hj}(\underline{x}))^2]$
    flipped_similarity $:= -\frac{1}{m}\Sigma_{k=0}^{m-1}[\Sigma_{\underline{x} \in D}\ (u_{hi}(\underline{x}) + u^k_{hj}(\underline{x}))^2]$
    $s_{ij} :=$ max(**similarity**,**flipped_similarity**)
    $f_{ij} := 0$ if **similarity** > **flipped_similarity**; $f_{ij} := 1$ otherwise
  $P :=$ permutation matrix that maximizes $\Sigma_{ij} s_{ij} p_{ij}$
  $\forall\ (i,j) \in \{1 \ldots n_h\} \times \{1 \ldots n_h\}$
    if $p_{ij} = 1$ and $f_{ij} = 1$ then flip the signs on hidden unit i in layer h of nn
  permute layer h of nn by P
return(nn,$\{nn^0, \ldots, nn^{m-1}\}$)
}

Likewise, the BNF algorithm can be extended as follows. The value of correlation(u_{hj},u^k_{hj}) is the correlation between corresponding thresholds on, and weights into and out of, hidden unit j in layer h of nn and hidden unit j in layer h of nn^k. (The correlation formula is given in the previous discussion of the BNF algorithm.)

BNF_set(nn, $\{nn^0, \ldots, nn^{m-1}\}$, D)
{
matrix S, P
int h, i, j

$\forall\ h \in \{1 \ldots L-1\}$
  $\forall\ (i,j) \in \{1 \ldots n_h\} \times \{1 \ldots n_h\}$
    $s_{ij} := -\frac{1}{m}\Sigma_{k=0}^{m-1}[\Sigma_{\underline{x} \in D}||nn_{hi}(\underline{x}) - nn^k_{hj}(\underline{x})||^2]$
  $P :=$ permutation matrix that maximizes $\Sigma_{ij} s_{ij} p_{ij}$
  permute layer h of nn by P
  $\forall\ j \in \{1 \ldots n_h\}$
    if $\frac{1}{m}\Sigma_{k=0}^{m-1}$ correlation(u_{hj},u^k_{hj}) ¡ 0 then flip the signs on u_{hj}
return(nn, $\{nn^0, \ldots, nn^{m-1}\}$)

}

These algorithms can be used in various ways to match all of the networks $nn^0, \ldots, nn^{n-1}$. Define match_set(nn, $\{nn^0, \ldots, nn^{m-1}\}$) to be a procedure that rearranges network nn to match it to networks $nn^0, \ldots, nn^{m-1}$. We can collect matched networks one at a time in the following fashion.

- match_set(nn^1,$\{nn^0\}$)
- match_set(nn^2,$\{nn^0, nn^1\}$)
- match_set(nn^3,$\{nn^0, nn^1, nn^2\}$)

⋮

-
- match_set(nn^{n-1},$\{nn^0, \ldots, nn^{n-2}\}$)

If there are multiple processes, then we can perform several matchings at once. For example, we can extend the fan-out procedure so that new networks are matched to the whole set of previously matched networks instead of to a single network.

- match_set(nn^0,$\{nn^{\frac{n}{2}}\}$)
- match_set($nn^{\frac{n}{4}}$,$\{nn^0, nn^{\frac{n}{2}}\}$), ($nn^{\frac{3n}{4}}$,$\{nn^0, nn^{\frac{n}{2}}\}$)

⋮

-
- match_set(nn^1,$\{nn^0, nn^2, \ldots, nn^{n-2}\}$), ..., match_set($nn^{n-1}$,$\{nn^0, nn^2, \ldots, nn^{n-2}\}$)

9.5 Proportioning Several Networks

In this section, we assume that networks $nn^0, \ldots, nn^{n-1}$ have been matched, and we explore methods to find a convex combination of the networks

$$nn(\underline{\theta}) \equiv \theta_0 nn^0 + \ldots + \theta_{n-1} nn^{n-1}, \Sigma_{i=0}^{n-1} \theta_i = 1 \tag{9.2}$$

that has low out-of-sample error.

9.5.1 Iterated Pairwise Proportioning

As with matching, several networks can be proportioned by iterating any of the pairwise proportioning algorithms presented earlier. Define proportion(nn^a,nn^b) to be a function that returns a weight-by-weight convex combination of nn^a and nn^b.

Accumulation Proportioning

We can proportion networks $nn^0, \ldots, nn^{n-1}$ by proportioning the first two networks, then proportioning the third network with the result of proportioning the first two, etc.

- $nn^{0,1}$:= proportion(nn^0,nn^1)
- $nn^{0,1,2}$:= proportion($nn^{0,1}$,nn^2)
- $nn^{0,1,2,3}$:= proportion($nn^{0,1,2}$,nn^2)

⋮

-
- $nn^{0,\ldots,n-1}$:= proportion($nn^{0,\ldots,n-2}$,nn^{n-1})

This method overemphasizes the later networks. Thus, it may be best to add the higher error networks to the mix earlier and save the lower error networks for the later proportionings.

Fan-In Proportioning

As with pairwise combination, we can achieve a balanced proportioning by fan-in. The networks are paired off, and each network is proportioned with its partner. Then the networks resulting from the partnerships are paired off and proportioned. The process is repeated until a single network results.

- $nn^{0,1}$:= combine(nn^0,nn^1), ..., $nn^{n-2,n-1}$:= combine(nn^{n-2},nn^{n-1})
- $nn^{0,3}$:= combine($nn^{0,1}$,$nn^{2,3}$), ..., $nn^{n-4,n-1}$:= combine($nn^{n-4,n-3}$,$nn^{n-2,n-1}$)

⋮

-
- $nn^{0,\ldots,n-1}$:= combine($nn^{0,\frac{n}{2}-1}$,$nn^{\frac{n}{2},n-1}$)

It is an open question whether it is better to pair together networks with similar error levels, or to pair low error networks with high error networks, as in a tournament in which higher-ranked teams play lower-ranked teams in the initial rounds.

9.5.2 Gradient Descent Proportioning

Network training is a search over a space with a number of dimensions equal to the number of weights in the network architecture. Gradient descent is a search method in which every step is chosen from a neighborhood with the

same dimension as the search space. Proportioning networks $nn^0, \ldots, nn^{n-1}$ to form $nn(\underline{\theta})$ is a search over a space with n dimensions. Pairwise proportioning searches a single dimension, so iterated pairwise proportioning methods search in one dimension at a time. In this section, we develop methods to search by selecting steps from a neighborhood with dimension n.

Proportioning can be performed by gradient descent. In network training, we minimize error by adjusting the network's weights. The update step is

$$\underline{w} := \underline{w} - \eta \frac{\partial E}{\partial \underline{w}} \tag{9.3}$$

where η is the stepsize multiplier and $\underline{w}$ is the weight vector.

In proportioning, we minimize error by adjusting the weights $\theta_0, \ldots, \theta_{n-1}$ of network $nn(\underline{\theta}) \equiv \theta_0 nn^0 + \ldots + \theta_{n-1} nn^{n-1}$. The update step is

$$\underline{\theta} := \underline{\theta} - \eta \frac{\partial E}{\partial \underline{\theta}} \tag{9.4}$$

The gradient of error with respect to each element of θ can be computed using the chain rule.

$$\frac{\partial E}{\partial \theta_i} = \frac{\partial E}{\partial nn(\underline{\theta})} \cdot \frac{\partial nn(\underline{\theta})}{\partial \theta_i} \tag{9.5}$$

Recall that

$$nn(\underline{\theta}) \equiv \theta_0 nn^0 + \ldots + \theta_{n-1} nn^{n-1} \tag{9.6}$$

So

$$\frac{\partial nn(\underline{\theta})}{\partial \theta_i} = nn^i \tag{9.7}$$

and hence

$$\frac{\partial E}{\partial \theta_i} = \frac{\partial E}{\partial nn(\underline{\theta})} \cdot nn^i \tag{9.8}$$

The gradient of the error with respect to $nn(\underline{\theta})$ can be computed by backpropogation. Thus, to compute the gradient $\frac{\partial E}{\partial \underline{\theta}}$, first compute $\frac{\partial E}{\partial nn(\underline{\theta})}$ by backpropogation, then take the dot product with each network nn^i to find $\frac{\partial E}{\partial \theta_i}$.

The update step above allows $nn(\underline{\theta})$ to be any linear combination of the networks. To limit the search space to linear combinations in which $\Sigma_{i=0}^{n-1} \theta_i = 1$, ensure that the condition holds initially, and project the update step into the hyperplane $\Sigma_{i=0}^{n-1} \frac{\partial E}{\partial \theta_i} = 0$, producing the following update step.

$$\underline{\theta} := \underline{\theta} - \eta [\frac{\partial E}{\partial \underline{\theta}} - \frac{\frac{\partial E}{\partial \underline{\theta}} \cdot \underline{1}}{\underline{1} \cdot \underline{1}} \underline{1}] \tag{9.9}$$

which is

$$\underline{\theta} := \underline{\theta} - \eta [\frac{\partial E}{\partial \underline{\theta}} - \frac{1}{n} \Sigma_{i=0}^{n-1} \frac{\partial E}{\partial \theta_i} \underline{1}] \tag{9.10}$$

To limit the search space to the convex combinations of the networks, we need to ensure that all elements of $\underline{\theta}$ are nonnegative. First, we must ensure that this condition holds initially. Then, we must alter update steps so that they will not cause any element of $\underline{\theta}$ to become negative.

If some element of $\underline{\theta}$ is positive, and the update would make the element negative, then we can scale the update to make the element zero. If some element θ_i is zero, and the update would make the element negative, then we project the update step into the subspace in which $\frac{\partial E}{\partial \theta_i} = 0$. This projection may cause the same problem for another element of $\underline{\theta}$, so after each projection we have to check for elements that would be made negative by the proposed update step. In essence, this projection is equivalent to performing gradient descent in the subspace in which θ_i is set to zero.

The following procedure produces an update that maintains the conditions $\Sigma_{i=0}^{n-1}\theta_i = 1$ and $\theta_i \geq 0 \forall i$. We begin by projecting the gradient into the subspace with element sum zero, to preserve the condition $\Sigma_{i=0}^{n-1}\theta_i = 1$. If this proposed update would make some zero component of $\underline{\theta}$ negative, then the gradient is projected into the subspace which does not include such components, and which has element sum zero. Then the new proposed update is checked to ensure that it will not make any zero component negative. Once we have an update that will not make any zero component negative, we check whether or not the update will make some positive component negative. If so, the update step is scaled so that it zeroes the positive component.

The vector $\underline{x}$ stores the update step. The set S stores the set of dimensions to be projected out of the update step.

1. $\underline{x} := \frac{\partial E}{\partial \underline{\theta}} - \frac{1}{n}\Sigma_{i=0}^{n-1}\frac{\partial E}{\partial \theta_i}\underline{1}$
2. If $\forall i \theta_i \neq 0$ or $x_i \leq 0$ then go to step 7
3. $S = \{i | \theta_i = 0 \text{ and } x_i > 0\}$
4. $\underline{x} := \frac{\partial E}{\partial \underline{\theta}} - \Sigma_{i \in S}\frac{\partial E}{\partial \theta_i}\underline{e_i}$ (Project out the elements of theta set to zero.)
5. $\underline{x} := \underline{x} - \frac{1}{n-|S|}[\Sigma_{i \notin S}\frac{\partial E}{\partial \theta_i}]\Sigma_{i \notin S}\underline{e_i}$ (Project into subspace with element sum zero.)
6. If $\exists i \notin S$ such that $\theta_i = 0$ and $x_i > 0$ then go to step 3
7. $\underline{x} := \eta \underline{x}$
8. $\forall i$ such that $\theta_i > 0$, if $\theta_i - x_i < 0$ then $\underline{x} := \frac{\theta_i}{x_i}\underline{x}$ (Scale to avoid making an element of $\underline{\theta}$ negative.)
9. $\underline{\theta} := \underline{\theta} - \underline{x}$

There is no compelling reason to limit the proportioning to convex combinations of the networks to be combined. Any linear combination of the networks is a valid network. If the best linear combination of the networks is convex, then the gradient descent procedure will return a convex combination (assuming that the search is successful.) Large elements in $\underline{\theta}$ generally indicate that the combination network's weights are larger (in absolute value) than the weights in the networks being combined. Negative entries in $\underline{\theta}$ correspond to networks that are subtracted from the combination.

The search over the whole space of linear combinations reaches a fixed point when the error gradient is zero, as in network training with gradient descent. The search over the subspace in which $\Sigma_{i=0}^{n-1}\theta_i = 1$ reaches a fixed point when all elements of the error gradient have the same value. At this point, the error gradient is orthogonal to the subspace. The search over convex combinations reaches a fixed point when the elements of the error gradient corresponding to positive elements of $\underline{\theta}$ are all equal, and the elements of the error gradient corresponding to zero elements of $\underline{\theta}$ are greater than or equal to the value of the other elements.

We assume that n, the number of networks to be proportioned, is less than the number of weights in each network. So proportioning is network training in an n-dimensional subspace of weight space. If n is small enough, then fancier minimization methods than gradient descent become feasible, including Newton's method and its variations [10].

Chapter 10

Combining Networks to Improve Generalization

Can we achieve better out-of-sample performance by training several networks and combining them than by training a single network? The tests detailed here show that combination improves generalization on a set of credit data.

I received the data from Joseph Sill, who in turn received the data from the machine learning database repository maintained by UC-Irvine. The following information about the credit data is drawn directly from Joseph Sill's paper on monotonicity hints [15].

"The credit card task is to predict whether or not an applicant will default. For each of the 690 applicant case histories, the database contains 690 features describing the applicant plus the class label indicating whether or not a default ultimately occurred. The meaning of the features is confidential for proprietary reasons. Only the 6 continuous features were used in the experiments reported here. 24 of the case histories had at least one feature missing. These examples were omitted, leaving 666 which were used in experiments. The two classes occur with almost equal frequency; the split is 55%-45%."

In each test the data is partitioned into an in-sample set and an out-of-sample set. Eight networks are trained on the in-sample data, and the networks are combined by fan-in. The out-of-sample errors are evaluated for the combined network and for the first trained network. The mean out-of-sample error rate for the single network is $22.58\% \pm 0.12\%$. The mean out-of-sample error rate for the combined network is $21.87\% \pm 0.12\%$. The combined network performs significantly better than the single network on out-of-sample data.

The estimates of the mean out-of-sample error rates are statistics over 1000 tests. The detailed procedure for each test is as follows:

- Randomly partition the 666 data examples into 555 in-sample examples and 111 out-of-sample examples.

- For each of eight 6-6-1 networks with random initial weights in [-0.1,0.1]:
  - Randomly partition the 555 in-sample examples into 444 training examples and 111 validation examples.
  - Train for 1000 epochs. Use sequential mode with random order of example presentation. Use the mean squared difference error function. Start with gradient descent multiplier 1.0. Adapt the stepsize after each epoch. If the training error decreases, then multiply the stepsize by 1.02. Otherwise, divide it by 2.
  - Evaluate the error on the validation data after each epoch. Return the network with minimum validation error.
- Combine the eight trained networks. Use fan-in combination with HUF matching, and use hybrid proportioning with five grid points and five levels of binary search. Use the in-sample data as the matching data.
- Evaluate the hard-classification error rate over the out-of-sample data for the following:
  1. the first of the eight trained networks
  2. the network with minimum hard-classification error rate on its validation data set (among the eight trained networks)
  3. the result of voting among the eight trained networks (If the vote is a tie, then the network outputs are averaged to determine the decision.)
  4. the combined network
  5. the network with the least out-of-sample error among the eight trained networks

The error on the first network reflects the effectiveness of training a single network. The out-of-sample error rate on the network with minimum validation error tests the result of training eight networks and choosing the one with the least validation error. This statistic is included to test whether the combination procedure is any more effective than training a set of networks and choosing one of them in this reasonable way. The voting error rate is included to compare an output fusion scheme against combination. The minimum out-of-sample error rate is included as a rough baseline to indicate the limit of classification ability imposed by our training method and our network architecture. The statistic is "dishonest", not only in the sense that it uses out-of-sample data to choose the classifier, but even more so because the same out-of-sample data used to select it is also used to test it.

The estimates of the mean out-of-sample error rates are shown in table 10. The single network performs worst. Choosing a trained network by minimum validation error improves the performance. Voting improves the performance even more. Combination yields better results than voting, but the difference is too small to be statistically significant.

classifier	mean error rate (%)
single network	22.58 ± 0.12
min. validation error network	22.34 ± 0.12
voting	21.93 ± 0.12
combined network	21.87 ± 0.12
min. out-of-sample error network	19.91 ± 0.11

Table 10.1: Mean out-of-sample error rate estimates for various classifiers on credit data.

Chapter 11

Multicomputer Implementations of Algorithms to Combine Several Networks

This chapter develops multicomputer implementations of some of the algorithms developed in the previous chapter. The notation used earlier to describe pairwise matching and proportioning algorithms is used in this chapter as well. Previously, it was assumed that there were two processes, with identifiers 0 and 1. In this chapter, it is assumed that there are n processes, labelled with identifiers $0, \ldots, n-1$. As before, the initial assumption is that the in-sample data is partitioned among the processes. Later, the duplication of data among processes is considered.

Some of the functions developed for pairwise combination are used in this chapter. To use the functions defined earlier in this new setting, their first lines should be changed from $p \in \{0, 1\}$ to $p \in \{0, \ldots, n-1\}$. In all other respects, the earlier function definitions are valid in this chapter.

This chapter begins with a high-level function which supplies a framework for the implementation of training and combination algorithms. A training method is developed in which the networks rotate among the processes to gain exposure to a variety of data. Two combination algorithms are implemented. The first algorithm uses repeated pairwise combinations to combine several networks. The second algorithm matches the networks, then proportions the matched networks by gradient descent on a linear combination of the matched networks. The chapter also contains the results of multicomputer tests designed to explore several aspects of the training and combination algorithms.

11.1 A Framework for Combining Several Networks

The following high-level function implements the training scheme with network combination illustrated in Figure 1.2 over n processes. In each interval, each process trains its network for several epochs, then the trained networks are combined, and, in each process, the combined network is trained further in the next interval.

```
p ∈ {0,...,n − 1}
net train_with_net_combination(net nn; data D; integer intervals, epochs; real η)
{
for i := 1 to intervals
    nn := train(nn, D, epochs, η);
    nn := combine(nn, D);
return nn;
}
```

The function train(nn, D, epochs, η) trains network nn on data set D for the specified number of epochs, using sequential mode backpropogation training with multiplier η. In each epoch, each example is presented to the network once, and the examples are presented to the network in random order. This is the basic procedure; a number of variations are possible. One variation, orbit training, is explored later in this chapter.

Training each network on only the data in its own process may cause overtraining, especially if each process contains very little data compared to the number of weights in the network. Intermittedly rotating the networks among the processes during the training phase exposes each network to a variety of training data. This process is called orbit training. In the following function, each process begins with its own network, trains it for a number of passes, then sends it to the process with the next higher identifier. After n train-and-send iterations, each network returns to its original process, completing an orbit.

```
p ∈ {0,...,n − 1}
net orbit_train(nn, D, orbits, passes, η)
{
for i := 1 to orbits
    for j := 0 to n − 1
        nn := train(nn, D, passes, η);
        send nn to (p + 1)mod n;
        receive nn from (p − 1)mod n;
return nn;
}
```

Orbit training with a single pass in each process achieves training that is similar to duplicating the entire training data set in each process and training networks separately in their processes. Orbit training requires more time, since the networks must be transported among the processes, but orbit training requires less memory in each process.

There is a slight difference in the training that occurs under the two regimes. With the training data duplicated in every process, the data examples can be presented to the network in a completely different order in each epoch. With orbit training, the order of presentation can only be changed within each process. The tests presented later in this chapter show that this difference has no more than a minor effect on training.

If there is not enough memory to duplicate all training data in every process, but there is more than enough memory to partition the data among processes, then there can be some duplication of data among processes, and networks can be routed among subsets of the processes and still get exposed to the entire data set. For example, if each process has twice as much memory as necessary to partition the data among processes, then the data can be partitioned among one half of the processes and separately partitioned among the other half. Each half of the processes can rotate its networks among its processes. This speeds computation since it reduces the number of communications per epoch by half.

11.2 Iterated Pairwise Combination

The function `combine_pair` combines the networks in processes `p` and `q`, where process `p` is the process that calls the function, and process `q` is specified as a function parameter. The function performs the pairwise combination described and implemented in previous chapters. First, the networks in the two processes are distributed over the processes, leaving copies of both networks in each process. Next, in one process, the networks are swapped so that the variables `nn` and `nn'` refer to the same networks in both processes. Then the networks are matched and combined using functions presented in the earlier chapter on implementing pairwise combination.

```
p ∈ {0,...,n − 1}
net combine_pair(net nn; data D; integer q)
{
send nn to q;
receive nn' from q;

if p > q then swap(nn, nn');

nn := match_pair(nn, nn', D, q);
```

```
θ := proportion_grid_pair(nn, nn', D, q);

return (1 − θ)nn + (θ)nn';
}
```

A fan-in communication pattern is used to combine several networks with iterated pairwise combination (see Figure 11.1.) Processes are paired off – 0 and 1, 2 and 3, 4 and 5, etc. In each pair of processes, the networks are combined, and the combined network replaces the originals in both processes. So the network in process 0 is a combination of the networks originally in processes 0 and 1, the network in process 2 is a combination of the networks originally in processes 2 and 3, etc. Next, the even processes are paired off – 0 and 2, 4 and 6, etc. Once again, in each pair of processes, the networks are combined, and the combined network replaces the originals. Now the network in process 0 is a combination of the networks originally in processes 0, 1, 2, and 3; the network in process 4 is a combination of the networks originally in processes 4, 5, 6, and 7. The process is iterated by pairing off processes with identifiers divisible by four – 0 and 4, 8 and 12, etc. Pairs are combined once again. The process is repeated until the networks in processes 0 and $\frac{n}{2}$ are combined. The resulting network is the culmination of combinations involving each of the original networks.

The active processes in layer m of the fan-in tree are the processes with identifers p that are multiples of 2^{m-1}, e.g. the first layer of fan-in involves all networks, and the second layer involves only processes with even identifers (see Figure 11.1.) The role of a process in fan-in is determined by the binary representation of its identifier p. Let k be the position (from the right) of the least significant bit with value one in the binary representation of p. Each process (other than process 0) participates in k combinations, successively combining its networks with those in processes $p + 2^0$, $p + 2^1$, ..., $p + 2^{k-2}$ and then process $p - 2^{k-1}$.

For example, process 6 has $k = 2$ since the binary representation of 6 is 110. Observe, in Figure 11.1, that process 6 participates in 2 combinations, first with process $6 + 2^0 = 7$, then with process $6 - 2^1 = 4$.

The function `last_one_bit` returns the location of the last bit with value one (counting from the left) in the binary representation of the process' identifier p. (If the process identifier is 0, then there is no one bit. In this case, the function returns $\log_2 n + 1$.)

```
p ∈ {0, ..., n − 1}
integer last_one_bit()
{
d := 1;
while p mod 2^d = 0 and 2^d ≤ n do d := d + 1;
return d;
```

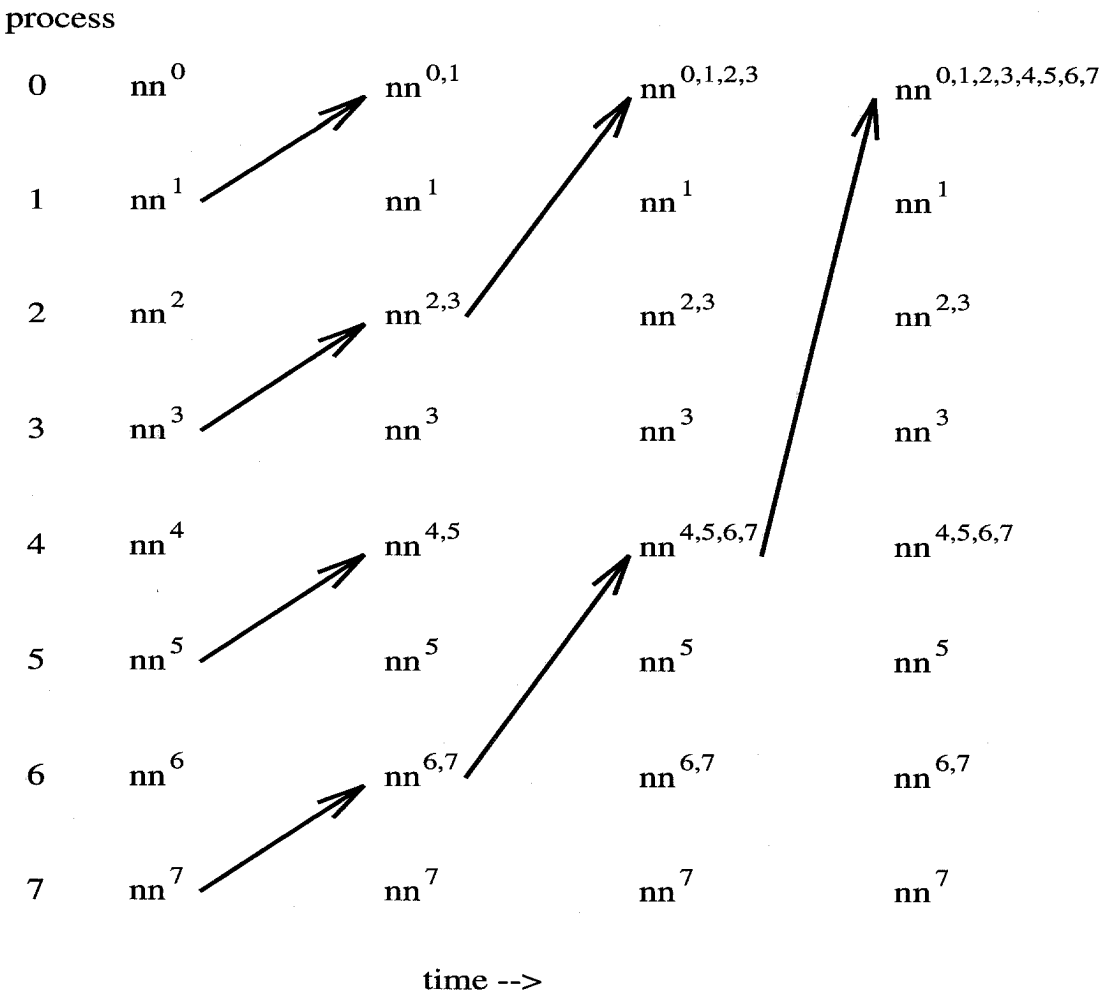

Figure 11.1: The message pattern for fan-in combination.

```
}

p ∈ {0, ..., n − 1}
net fan_in_combine(net nn; data D)
{
d := 0;
k := last_one_bit();

while d ≤ k − 2 do
    nn := combine_pair(nn, D, p + 2^d);
    d := d + 1;

if p ≠ 0 then nn := combine_pair(nn, D, p − 2^(k−1));
return nn;
}
```

After fan-in combination, the full combination network resides in process 0, but not in most of the other processes. A fan-out pattern of communications distributes the full combination network from process 0 to all processes. The fan-out pattern is the reverse of the fan-in pattern. First, process 0 sends the full combination network to process $\frac{n}{2}$. Then process 0 sends the network to process $\frac{n}{4}$ while process $\frac{n}{2}$ sends the network to process $\frac{3n}{4}$. In the final step, process 0 sends the network to process 1, process 2 sends the network to process 3, process 4 sends the network to process 5, etc.

Let k be the location of the last one bit. For processes other than process 0, k is the number of communications that the process performs in the course of a fan-out. First, the process receives the network from process $p - 2^{k-1}$. Then the process sends the network to processes $p + 2^{k-2}$, $p + 2^{k-3}$, ..., $p + 2^0$.

```
p ∈ {0, ..., n − 1}
net fan_out_net(net nn)
{
k := last_one_bit();
d := k − 2;

if p ≠ 0 then receive nn from p − 2^(k−1);

while d ≥ 0 do
    send nn to p + 2^d;
    d := d − 1;

return nn;
}
```

11.2.1 Fan-In Combination Tests

Figures 11.2 and 11.3 show the results of tests to determine the rate of error reduction over the course of fan-in combination. In each test, sixteen networks were trained and then combined by fan-in. At each level of fan-in, the test error was computed for each network, and the minimum test error was recorded. (Level zero contains the initial sixteen networks, level one contains eight networks, and so on. Level four contains the single network that is the culmination of fan-in combination.) Figure 11.2 shows the average and standard deviation of the minimum test error for the networks at each level of combination. On average, the minimum test error decreases over the course of fan-in combination. For each level after the initial one, the difference between the minimum test error in the level and the minimum test error in the previous level was recorded. Figure 11.3 shows the average and standard deviation of the difference in minimum test error between each level and the previous level. On average, each level of fan-in combination reduces the minimum test error.

In each run, a random problem is generated using the twin networks procedure. Each network has 10 input units, 10 hidden units, and 10 output units. The teacher network weights are drawn randomly from $[-10, 10]$. The student network initial weights are drawn from $[-0.1, 0.1]$. The weights of each student network are drawn independently.

Each run uses a test data set with 2200 examples (10 examples per weight.) The student network data sets are created independently. Each data set has 440 examples (2 examples per weight.) Each student trains for 1000 sequential mode epochs, with backpropogation multiplier 0.01 and with random order of example presentation in each epoch. The trained student networks are combined by fan-in, using HUF matching and hybrid proportioning with 5 grid partitions and 5 levels of binary search.

11.2.2 Multicomputer Tests

The results in this section are for multicomputer test runs of variations of the following procedure.

```
p ∈ {0,...,n − 1}
net train_with_fan_in_combination(net nn; data D; integer intervals, epochs; real η)
{
for i := 1 to intervals
    nn := train(nn, D, epochs, η);
    nn := fan_in_combine(nn, D);
return nn;
}
```

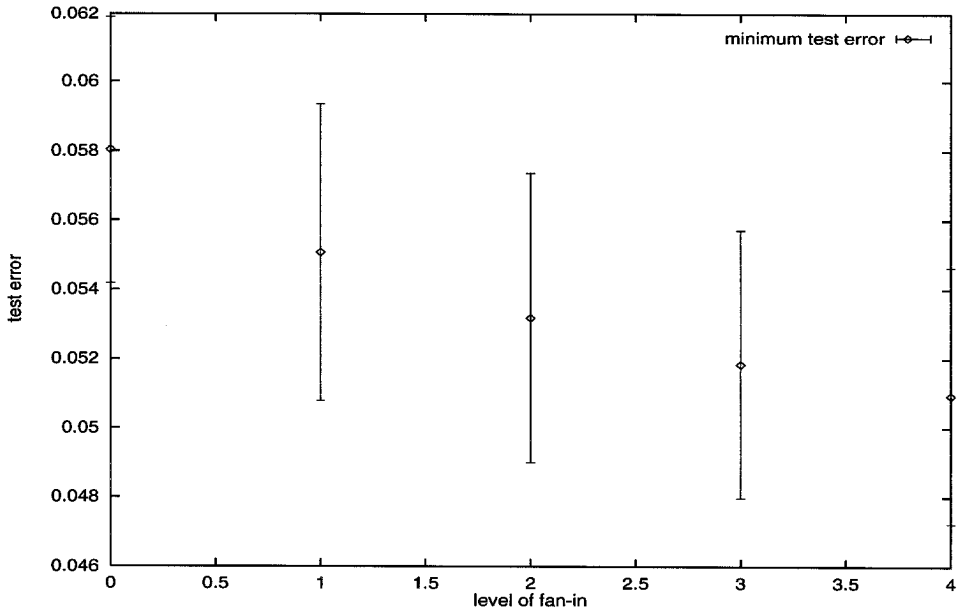

Figure 11.2: The minimum test error decreases through the levels of fan-in combination.

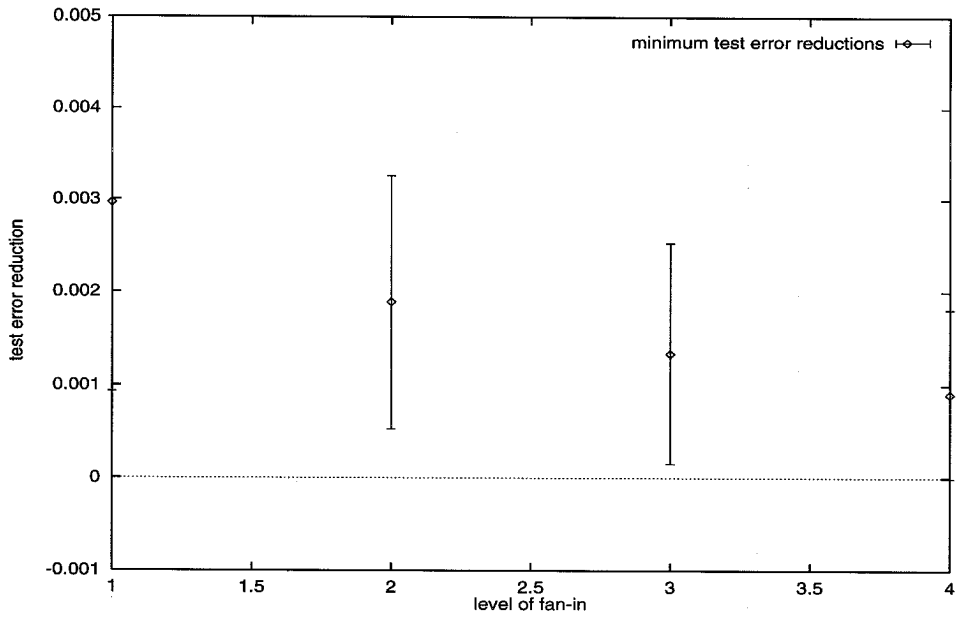

Figure 11.3: On average, each level of fan-in combination reduces the minimum test error.

Each test consists of 10 training intervals, with 100 training epochs in each interval. The data sets are generated using the twin networks framework, with 10-10-10 networks. The test runs each use the same training and test data sets, generated by a teacher network with weights drawn uniformly at random from $[-10, 10]$. The training data set has 8 examples per network parameter (1760 examples), and the test data set has 10 examples per network parameter (2200 examples). Each test run uses the same initial network weights, drawn uniformly at random from $[-0.1, 0.1]$.

The backpropogation stepsize multiplier η is adapted during training. Initially, $\eta = 1.0$. After each training epoch, the training error is measured. If the the training error decreased during the epoch, then η is multiplied by 1.2; if the training error increased during the epoch, then η is multiplied by 0.5. Each process adjusts its multiplier independently, so the value of η is free to vary among the processes.

In every case, the error used to adapt the stepsize is the error of the network being trained over the data presented to the network during the training epoch. For example, when the training data is partitioned among the processes, and each process trains its own network, each process adjusts its stepsize according to the error of the process' network over the process' subset of the training data.

Sequential mode training is used. In each epoch, the training examples are presented in a random order. Each pairwise combination uses HUF matching and grid proportioning, with grid spacing 0.1.

In each run, the errors are recorded for the network in process 0. After each training epoch and after each combination, the errors are computed over the training data, over the test data, and over the subset of training data in process 0. These errors are called the training, test, and in-process errors, respectively.

Partitioned Data

In these tests, the training data is partitioned among the processes. In each interval, each process trains its network for 100 epochs on its subset of the training data.

Figure 11.4 shows the errors for a run on 8 processes. During each interval, the in-process error decreases through training on the in-process data. However, the training and test errors increase during training because each process has only 220 examples (one example per network weight) of training data, so overtraining occurs. Combination increases the in-process error and decreases the training and test errors. The training and test errors of the combined network decrease with each interval, so the training procedure is successful.

Figure 11.5 shows the errors for a run on 32 processes. Each process has only 55 examples (one quarter as many training examples as the number of network weights.) So the increases in training and test error during the training phase

are even more pronounced than with 8 processes. Nonetheless, the training and test errors after network combination decrease with every interval, so learning occurs.

Figure 11.6 shows the errors for a run on 2 processes. Each process has 880 examples (four examples per network weight.) In this case, the training, test, and in-process errors remain close throughout training. Figure 11.7 is a close-up of Figure 11.6 in which the first 100 training epochs are eliminated, and a smaller error range is displayed. This closer view reveals behavior similar to the runs with 8 and 32 processes. However, in this case the training and test errors increase only at the beginning of each training phase. Eventually, training causes all of the errors to decrease. This is because each process has enough to data to train its network independently.

Duplicated Data

In this run, the training data is duplicated among 8 processes. In each interval, each process trains its network on all training data. So each process is capable of training its own network without overtraining, and combination does not combine the effects of training on different data sets. Instead, combination combines the effects of different orders of presentation of training examples to the networks in each process.

Figure 11.8 shows that this training process is quite successful. Training and test errors decrease rapidly, and no overtraining occurs. (Training and in-process errors are the same, since each process contains all training data.) Figure 11.9 is a close-up of Figure 11.8. The first 100 training epochs are eliminated, and a smaller error range is displayed. This closer view reveals that both training and test errors decrease as a result of both training and combination.

Orbit Training

In this run, orbit training is used. The training data is partitioned among the 8 processes, but, in each training epoch, each network is exposed to every training example. This is accomplished by rotating the networks through the processes. In each process, the network trains for a single pass on the subset of the training data found in the process. The backpropogation stepsize η is adapted for each network using the error over all training data. Also, each network carries its stepsize η with it on its rotation through the processes.

Although the algorithm is different, the effect is much the same as training with the entire data set duplicated in each process. The only difference is that the order of presentation of examples is entirely random for the duplicated data case, but the order of presentation of examples is only random within process data subsets in the orbit training case.

Figure 11.10 shows successful learning. The close-up, Figure 11.11, is quite similar to the close-up of training with duplicated data, Figure 11.9.

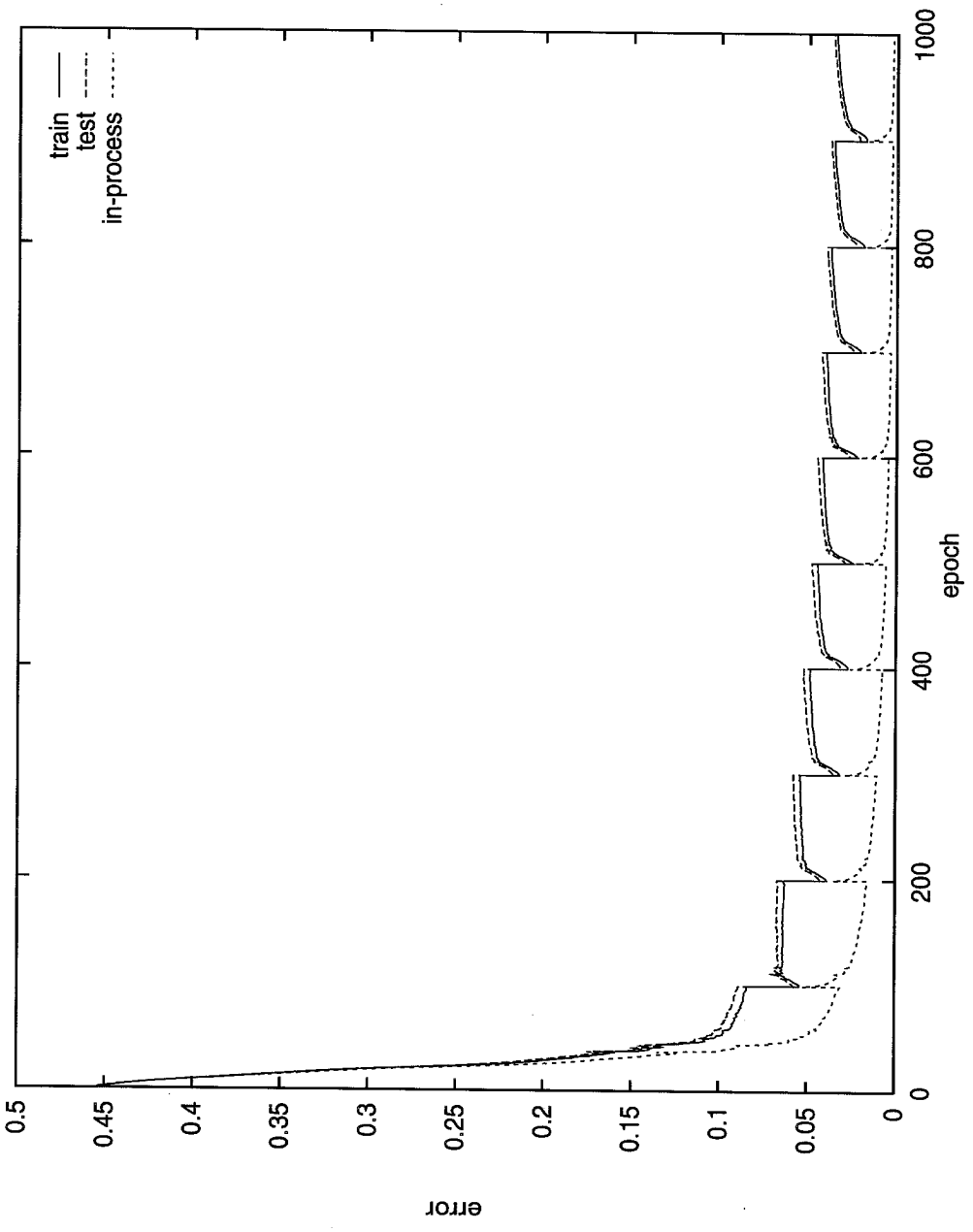

Figure 11.4: Fan-in Combination – 8 Processes – Partitioned Data

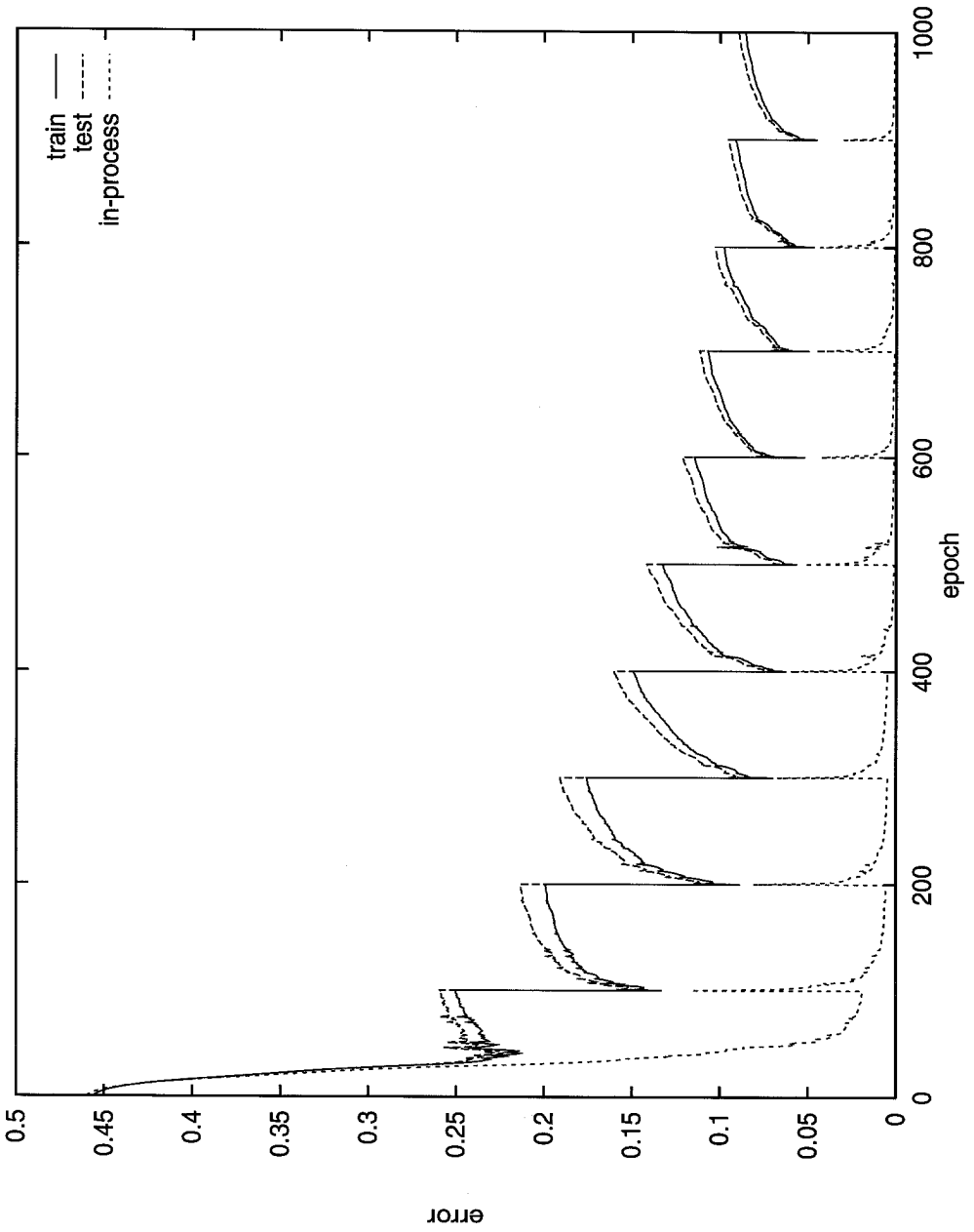

Figure 11.5: Fan-in Combination – 32 Processes – Partitioned Data

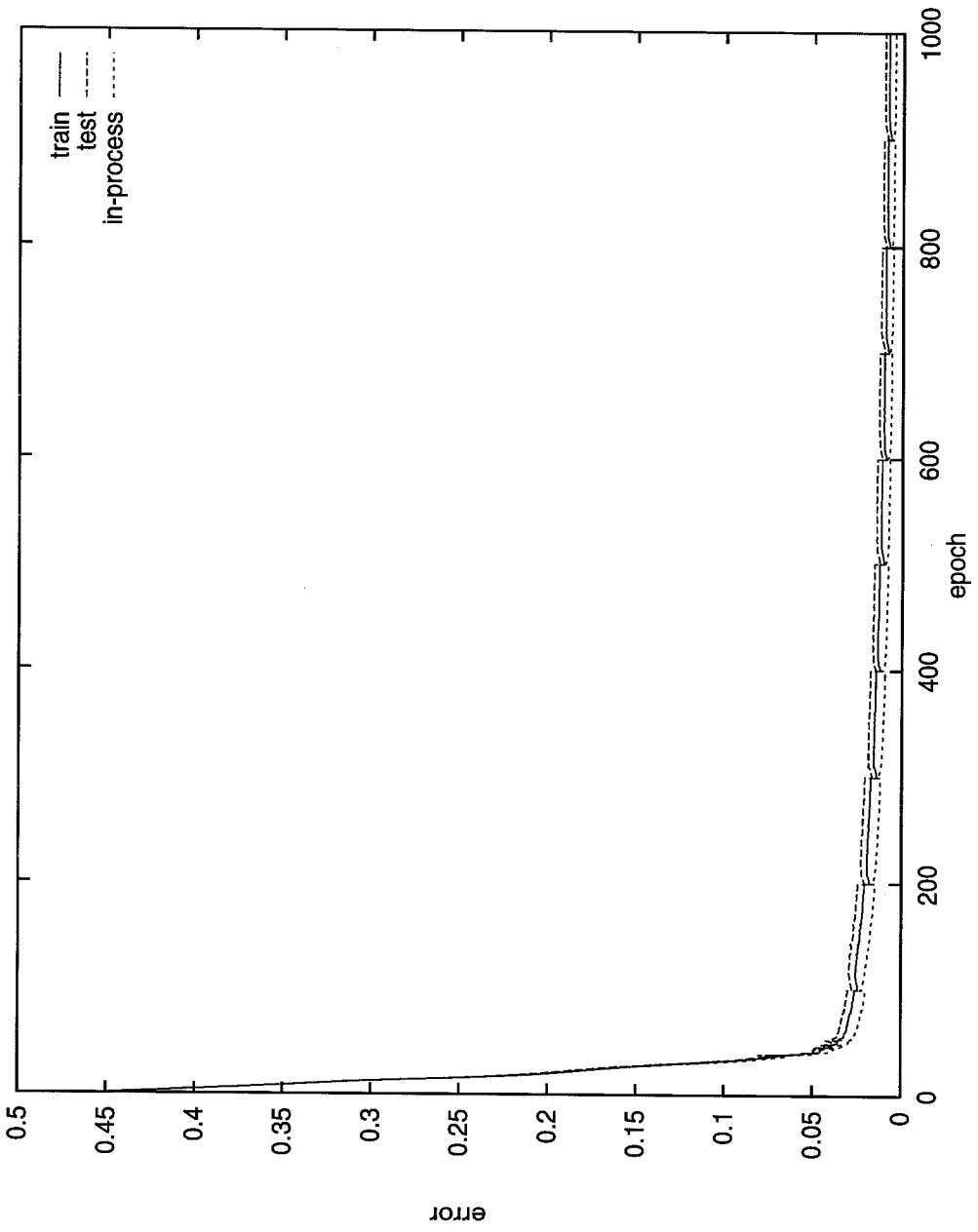

Figure 11.6: Fan-in Combination – 2 Processes – Partitioned Data

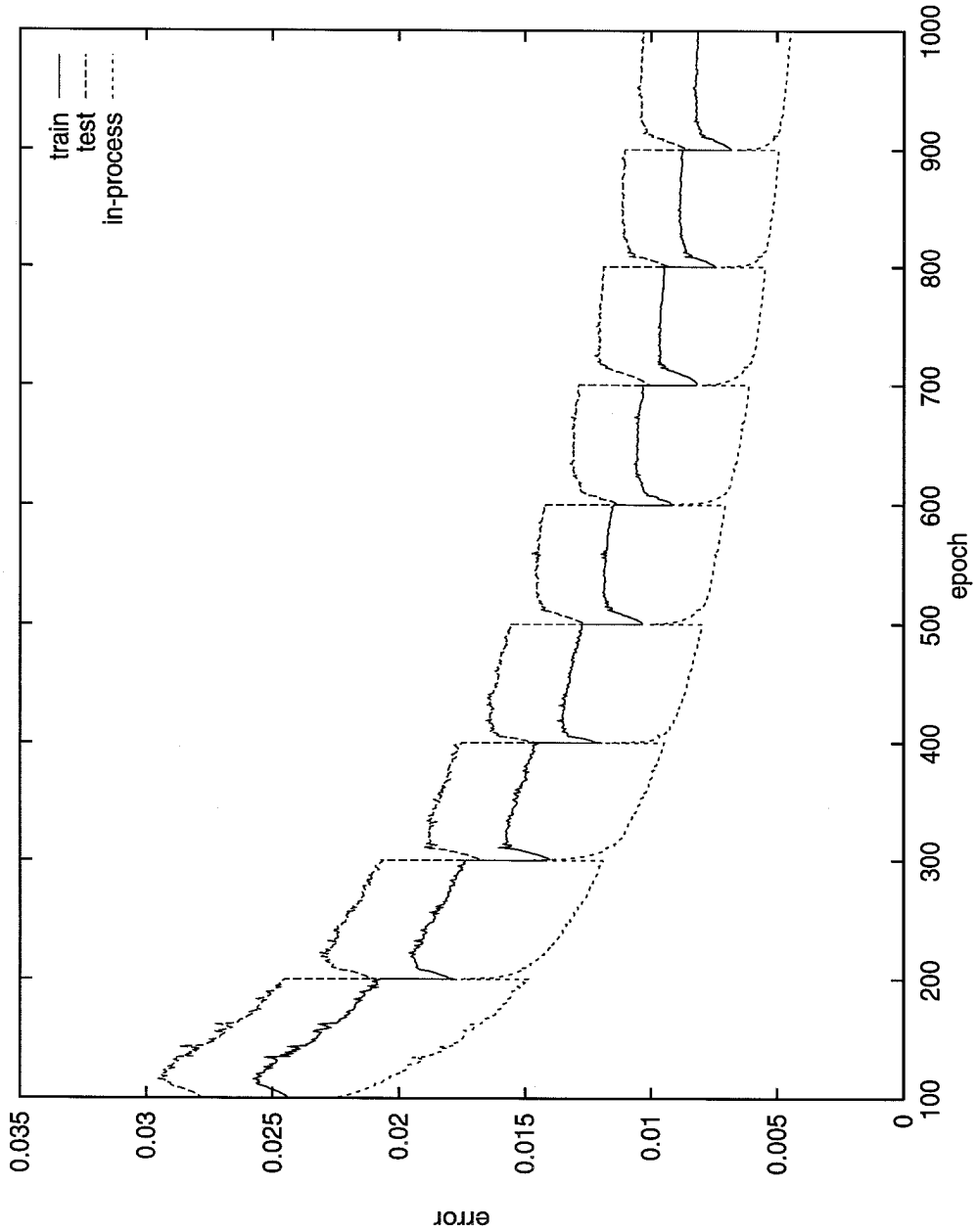

Figure 11.7: Detail of Fan-in Combination – 2 Processes – Partitioned Data

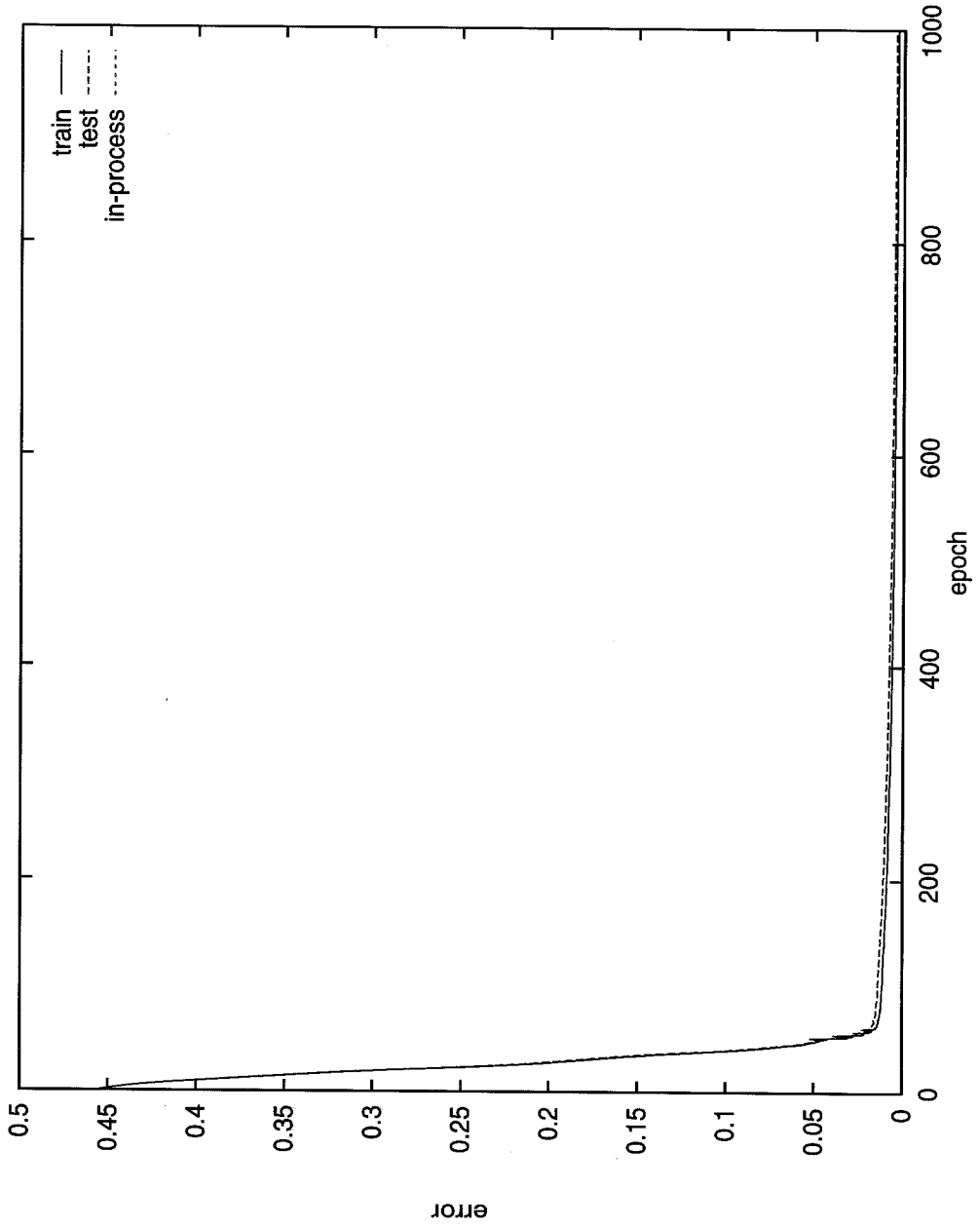

Figure 11.8: Fan-in Combination – 8 Processes – Duplicated Data

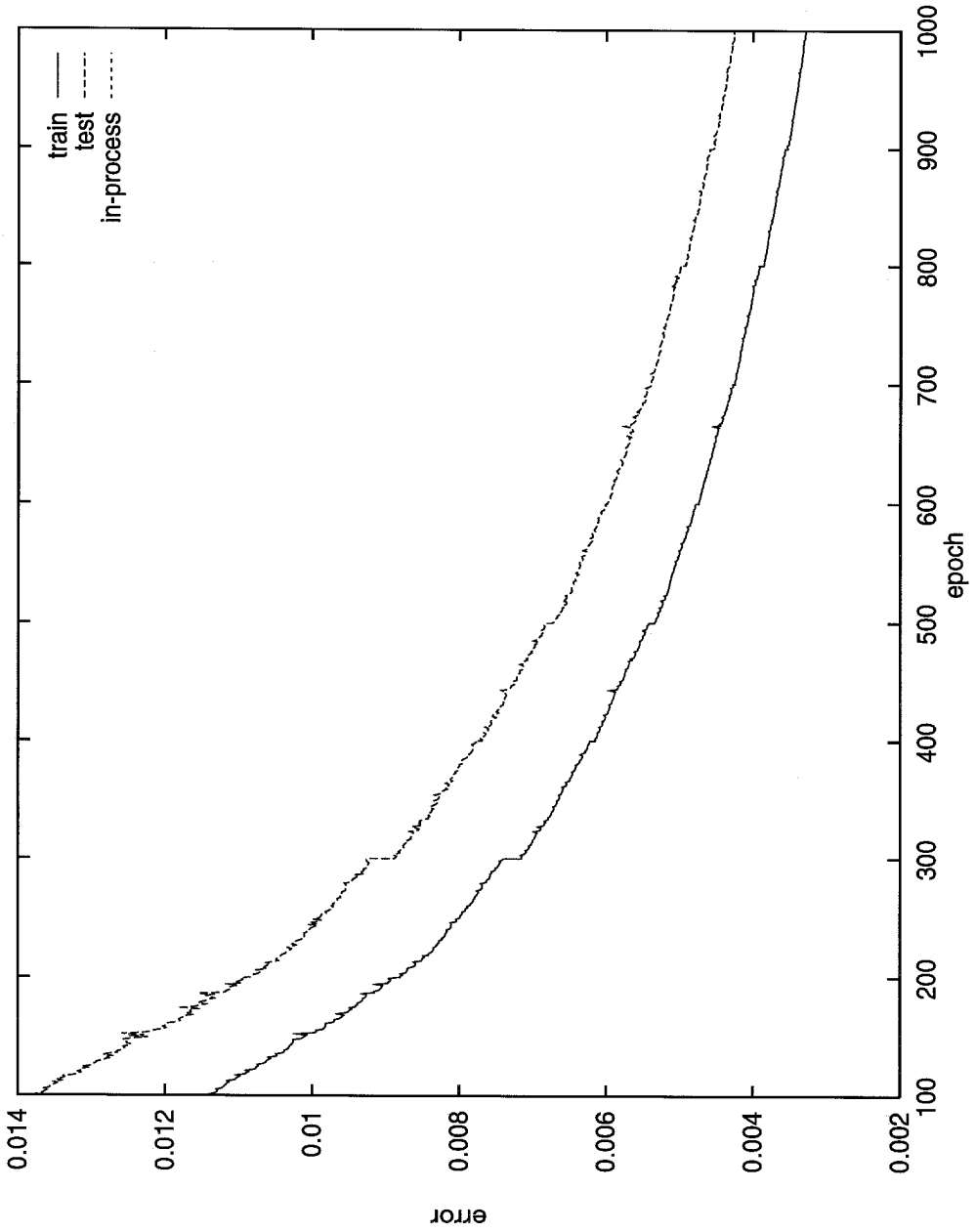

Figure 11.9: Detail of Fan-in Combination – 8 Processes – Duplicated Data

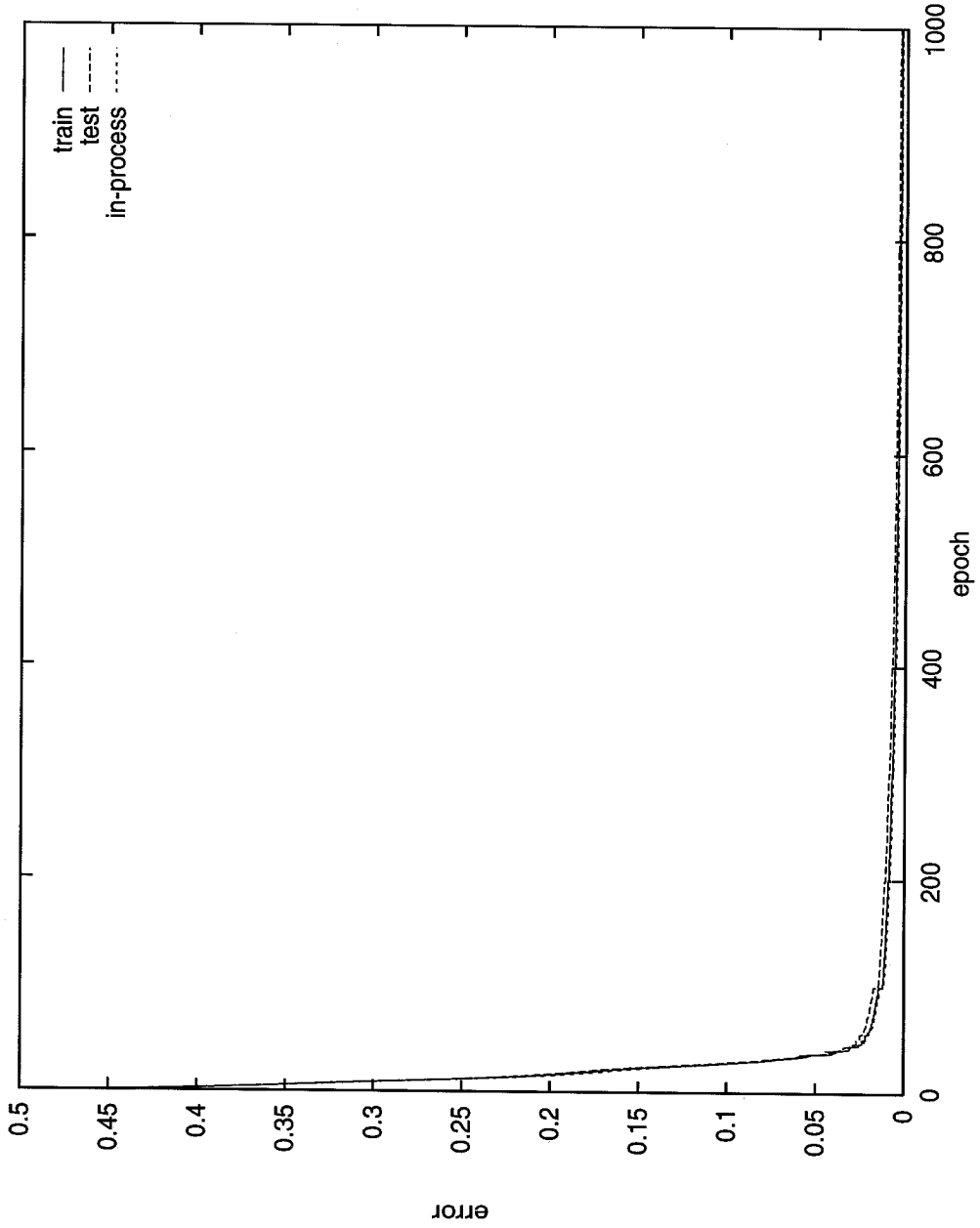

Figure 11.10: Fan-in Combination – 8 Processes – Orbit Training

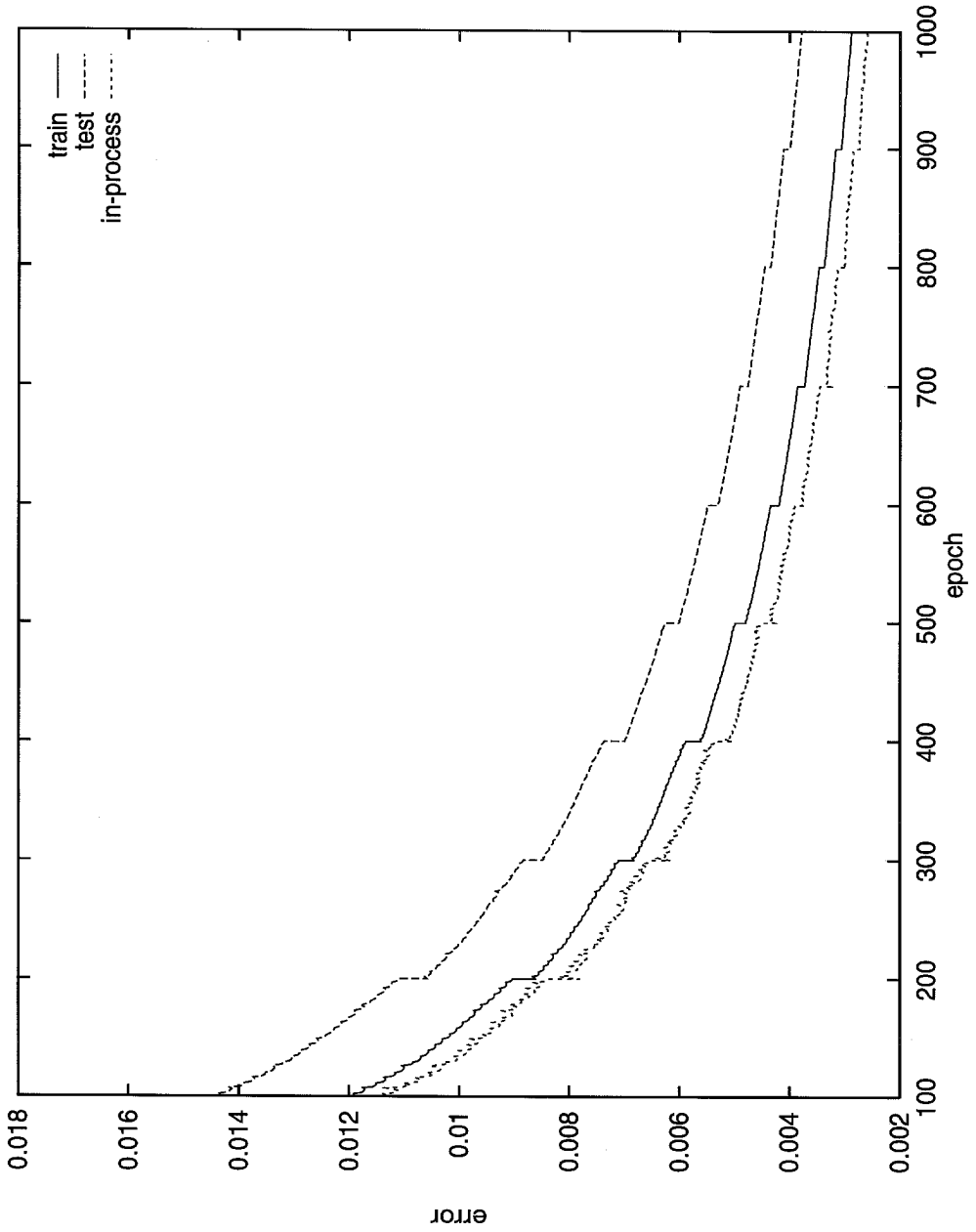

Figure 11.11: Detail of Fan-in Combination – 8 Processes – Orbit Training

11.3 Combination by Fan-Out Matching and Gradient Descent Proportioning

11.3.1 Implementation of Fan-Out Matching

Fan-out matching uses repeated pairwise matching to match several networks. The network in process 0 is the template. The networks in each of the other processes are matched to the network in process 0, either directly or through a chain of pairwise matchings. In the first layer of fan-out, the network in process $\frac{n}{2}$ is rearranged to match it to the network in process 0. In the second layer, the network in process $\frac{n}{4}$ is matched to the network in process 0, and the network in process $\frac{3n}{4}$ is matched to the network in process $\frac{n}{2}$. In the final layer, each network in a process with an odd identifier is matched to a network in a process with an even identifier, i.e. the network in process 1 is matched to the network in process 0, the network in process 3 is matched to the network in process 2, etc.

The function `fan_out_match` has the same structure as the function `fan_out_net`, described previously. Let k be the location of the last one bit. For processes other than process 0, k is the number of matchings that the process performs in the course of the fan-out. First, the process rearranges its network to match it to the network in process $p - 2^{k-1}$. Then the process uses its network as a template to rearrange the networks in processes $p + 2^{k-2}$, $p + 2^{k-3}$, ..., $p + 2^0$.

```
p ∈ 0,...,n − 1
net fan_out_match(net nn; data D)
{
k := last_one_bit();
d := k − 2;

if p ≠ 0 then
    send nn to p − 2^(k−1);
    receive nn' from p − 2^(k−1);
    nn := match_pair(nn, nn', D, p − 2^(k−1));

while d ≥ 0 do
    send nn to p + 2^d;
    receive nn' from p + 2^d;
    nn' := match_pair(nn', nn, D, p + 2^d);
    d := d − 1;

return nn;
}
```

11.3.2 Network Distribution

After fan-out matching, each process contains only its own matched network. The implementation of gradient descent proportioning presented later in this chapter requires each process to contain copies of the matched networks from every process. This section details a method to distribute copies of all networks to all processes.

The function `distribute_nets` uses a communication pattern called recursive doubling, or the butterfly (see Figure 11.12.) The main loop uses the counter `d` to iterate over the bits in the binary representation of the process identifier `p`. The counter runs from 0 to $\log_2 n - 1$, referring to the bits from the least to the most significant.

Initially, `d` has value 0, and each list `L` in each process contains only the network trained and matched in the process. In the first iteration, each process with an even identifier sends its network to the processor with the next higher identifier, and each processor with an odd identifer sends its network to the process with the next lower identifier. Each process receives a network. If the process has an even identifier, then it receives a network from the next higher-numbered process, so it appends the received network to its own network to update the list `L`. On the other hand, if the process has an odd identifier, then it receives a network from the next lower-numbered process, so it appends its own network to the received network to update the list `L`. Now each process has an ordered list containing its own network and the network from the process with the identifier that has the same binary representation, except that the least significant bit is different.

When `d` is assigned a value at the beginning of an iteration, the ordered list `L` in each process contains the networks from all processes with identifiers that agree with `p` in all bits except for the `d` least significant bits. In the iteration, process `p` receives from process $\mathtt{p} \otimes 2^{\mathtt{d}}$ all of the networks from processes with identifiers that disagree with `p` in bit $d+1$ and agree with `p` in the more significant bits. After these networks are added to the ordered list, `L` contains the networks from all processes with identifiers that agree with `p` in all bits except for the $\mathtt{d}+1$ least significant bits. By induction, after the iteration with `d` assigned $log_2 n - 1$, the ordered list in each process contains the networks from all processes.

```
p ∈ {0, ..., n − 1}
net[] distribute_nets(net nn)
{
L := (nn);

for d := 0 to log2 n − 1
    send L to p ⊗ 2^d;
    receive L' from p ⊗ 2^d;
    if p < p ⊗ 2^d then L := append(L, L'); else L := append(L', L);
```

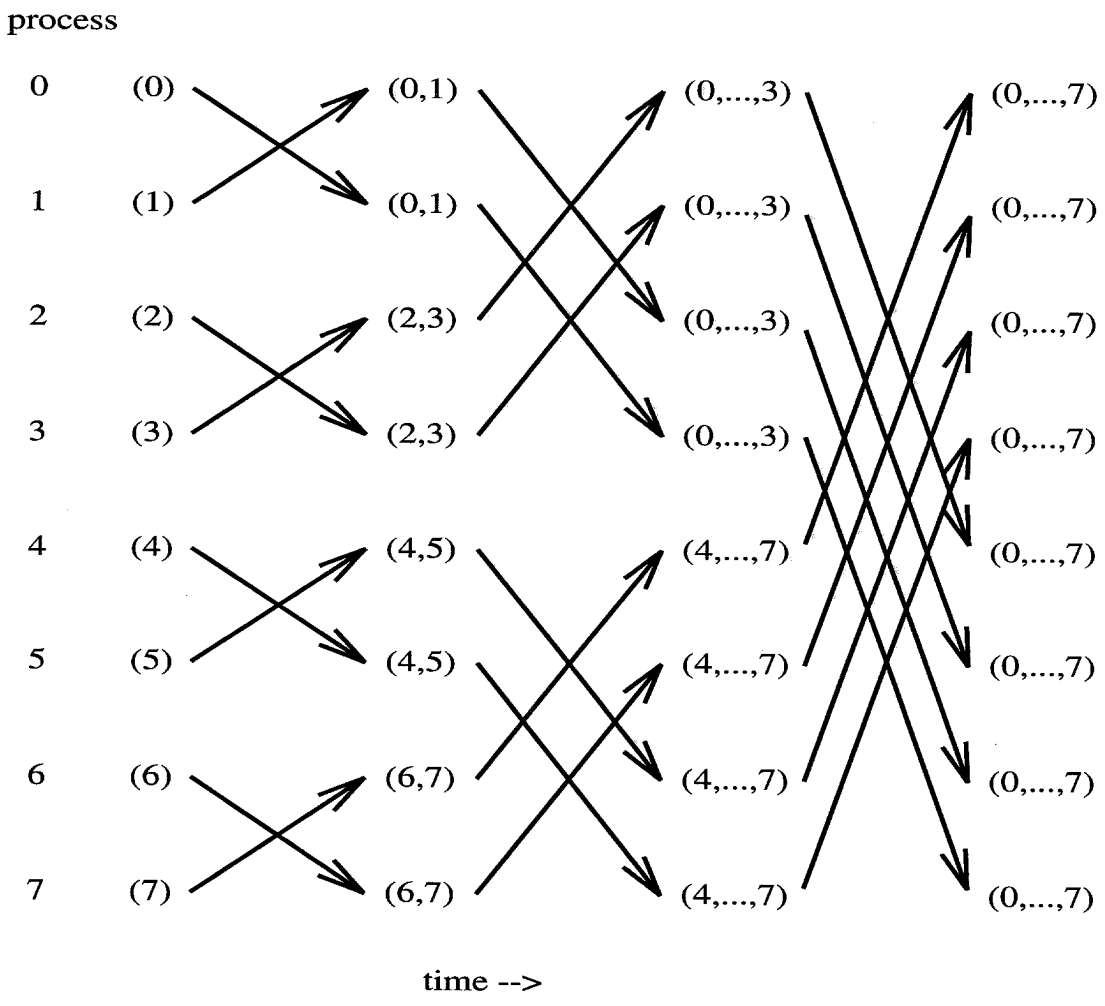

Figure 11.12: The message pattern to distribute networks by recursive doubling.

```
return L;
}
```

11.3.3 Gradient Descent Proportioning

This section details the implementation of the gradient descent proportioning algorithm developed in the previous chapter. The algorithm produces a linear weighting of several matched networks by descending the gradient of the error with respect to the weighting.

Support functions are developed to sum scalar values and network weight vectors over all processes. A function is developed to compute the gradient of the error with respect to a network over the data in all processes. Then a function is developed to implement gradient descent proportioning.

It is assumed that each process contains copies of all the matched networks and the data is partitioned among the processes. The algorithm searches the region of weight space spanned by the matched networks. No sum or sign constraints are imposed on the linear combination of networks.

The following function sums the value of a single real variable over all processes, leaving the sum in each process. The function uses recursive doubling, and it has the same structure as the function `distribute_nets`, which was described earlier. Instead of building a list as in `distribute_nets`, `sum` simply sums received values.

```
p ∈ {0,...,n − 1}
real sum(real x)
{
for d := 0 to log_2n − 1
    send x to p ⊗ 2^d;
    receive x' from p ⊗ 2^d;
    x := x + x';

return x;
}
```

The following function sums network weight vectors over all processes, leaving the sum in each process. It has the same structure as the function `sum` above. The only difference is that it sums vectors instead of real numbers.

```
p ∈ {0,...,n − 1}
net sum_net(net nn)
{
for d := 0 to log_2n − 1
```

```
    send nn to p ⊗ 2^d;
    receive nn' from p ⊗ 2^d;
    nn := nn + nn';

return nn;
}
```

The function grad_E_nn finds the gradient of the error with respect to a network over the data in all processes. First, backpropogation [11] is used to find the gradient over the data within each process. Then the function sum_net sums the gradient vectors over all processes.

```
p ∈ {0, ..., n − 1}
net grad_E_nn(net nn; data D)
{
grad := backprop(nn, D);
return sum_net(grad);
}
```

The function gradient_descent_step implements the gradient descent proportioning update step developed in the previous chapter. The function grad_E_nn computes the error gradient with respect to the current linear combination nnθ of matched networks nn[0...n − 1]. The dot product of this gradient with each network weight vector nn[i] is the partial derivative of the error with respect to θ[i]. The function updates the vector θ[0...n − 1] by subtracting a multiple of the error gradient, βgrad_E_θ.

```
p ∈ {0, ..., n − 1}
real[] gradient_descent_step(real θ[0...n − 1]; net nn[0...n − 1]; data D; real β)
{
nnθ := θ[0]nn[0] + ... + θ[n − 1]nn[n − 1];
grad_E_nnθ := grad_E_nn(nnθ, D);
for i := 0 to n − 1 grad_E_θ[i] := grad_E_nnθ · nn[i];
θ := θ − βgrad_E_θ;
return θ[0, ..., n − 1];
}
```

11.3.4 Putting it All Together

The following function combines the processes' networks with the separate matching and proportioning functions developed earlier in this chapter. Initially, each process has a trained network. The networks are matched by fan-out.

Next, the networks are distributed among the processes so that each process has copies of all of the networks. Then the networks are proportioned by gradient descent. The initial weighting among networks, $\theta[0 \ldots n-1] := (\frac{1}{n}, \ldots, \frac{1}{n})$, is a balanced linear combination of the networks. The weights $\theta[0 \ldots n-1]$ are adjusted by a number of gradient descent updates specified by the parameter steps, with the step size multiplier specified by the parameter β. The function returns the linear combination of matched networks $nn[0 \ldots n-1]$ specified by the value of $theta[0 \ldots n-1]$ after gradient descent.

```
p ∈ {0, ..., n − 1}
net combine(net nn; data D; integer steps; real β;)
{
nn := fan_out_match(nn, D);
nn[0 ... n − 1] := distribute_nets(nn);
θ[0 ... n − 1] := (1/n, ..., 1/n);
for i := 1 to steps
    θ[0 ... n − 1] := gradient_descent_step(θ[0 ... n − 1], nn[0 ... n − 1], D, β);
return θ[0]nn[0] + ... + θ[n − 1]nn[n − 1];
}
```

11.3.5 Multicomputer Tests

This section contains the results of multicomputer training runs using combination by fan-out matching and gradient descent proportioning. Each run executes a variation of the following procedure.

```
p ∈ {0, ..., n − 1}
net train_and_combine(net nn; data D; integer intervals, epochs; real η; integer steps; real
{
for i := 1 to intervals
    nn := train(nn, D, epochs, η);
    nn := combine(nn, D, steps, β);
return nn;
}
```

The framework used for the runs discussed here is the same as the framework used for the runs with fan-in combination. The initial network weights and the data sets are the same. The backpropogation stepsize multiplier is adjusted during training in the same manner as in the previous tests.

In these tests, the network proportioning gradient descent stepsize β is also adjusted adaptively. After each proportioning step, the error of the combined network is computed over all training data. If the error decreased, then β is

multiplied by 1.2. Otherwise, β is multiplied by 0.5. The stepsize β is initially 1.0. Each proportioning involves 10 gradient descent steps.

The training, test, and in-process errors are computed as in the previous tests. For these tests, we also compute the training and test errors after each gradient descent step in the proportioning procedure. The training error is the error of the combined network over all training data, and the test error is the error over all test data.

Partitioned Data

In this run, the training data is partitioned among 8 processes. In each interval, each process trains its network on its share of the training data. Then the trained networks are combined. Figure 11.13 shows the evolution of errors during training. The errors exhibit behavior similar to the fan-in combination run with 8 processes. In the training phase of each interval, the training and test errors increase, while the in-process error decreases. Then combination decreases the training and test errors. On an epoch-by-epoch basis, the combination procedure used in this run gives faster learning than the fan-in combination procedure.

Figure 11.14 shows the evolution of errors during proportioning. Each sequence of 10 proportioning steps corresponds to a single combination. The gradient descent proportioning steps tend to smoothly reduce training and test error.

Figure 11.15 is a close-up showing the error evolution over the course of both training epochs and proportioning steps. The training and test errors consist of sequences beginning with a dip, then increasing, followed by a sharp drop to the beginning of the next sequence. In each sequence, the descending portion of the dip corresponds to gradient descent proportioning, the rising portion corresponds to training, and the sharp drops indicate the error reduction achieved by matching the networks and taking the weight-by-weight average to begin gradient descent proportioning. The sharp drops indicate that the initial weight-by-weight average over the networks decreases the errors much more than the subsequent gradient descent steps. This indicates that weight-by-weight averaging over several networks is robust.

Duplicated Data

In this run, the the training data is duplicated in each process. The evolution of the errors over the course of the run is shown in Figure 11.16. Since each process has enough data to train its network without overtraining, the training and test errors decrease through both training and combination. As a result, the training procedure is quite successful.

Orbit Training

This run employs orbit training to present every training example to each network in each training epoch. The data is partitioned among 8 processes, and the networks rotate through the processes during each epoch, training for a single pass in each process. The effect is similar to training with duplicated data.

Figures 11.17 and 11.18 show that this is a successful training procedure. The training and test errors generally decrease through both training and combination. The only problem encountered is a result of the first combination. Look at the first 10 steps shown in Figure 11.18. Instead of decreasing smoothly, the errors jump about. Apparently, the initial gradient descent stepsize $\beta = 1.0$ is too large for effective proportioning. Although this reduces the effectiveness of the first combination, the stepsize is adjusted through adaptation, and the remaining combinations proceed smoothly.

11.4 Learning Speeds

Figure 11.19 plots the evolution of test error for the training methods discussed in previous sections of this chapter, as well as the standard method of training a network on a multicomputer – batch mode training with error gradients summed over processes every epoch. Each method is applied to the same problem, described previously. Each test run uses 8 processes and executes 1000 training epochs. Figure 11.20 is a close-up of Figure 11.19.

The standard training method executes batch mode training, so its training proceeds slowly but steadily. The other methods train in each process for 100 epochs between combinations, so their curves are punctuated by error reductions caused by combination.

Fan-in combination is weaker than combination with gradient descent proportioning. Each fan-in combination curve is higher than the corresponding curve for combination with gradient descent proportioning. This is no surprise – while both methods consist of a series of steps in weight space, the steps of fan-in combination are constrained along lines, while the steps of the other method are only constrained to lie in the subspace spanned by the networks being combined.

For both combination methods, duplicating the data in each process and orbit training with partitioned data yield lower test errors than training each network on the data partition in a single process. This is as expected, since the first two methods expose every network to the full data set in each epoch of training. However, there is a tradeoff involving time – duplicated data and orbit training each require 8 times the training time per epoch as training on partitioned data (since there are 8 processes.) For orbit training, there is also the added expense of communicating to rotate the networks through the processes. Neither the duplicated data method nor the orbit training method executes

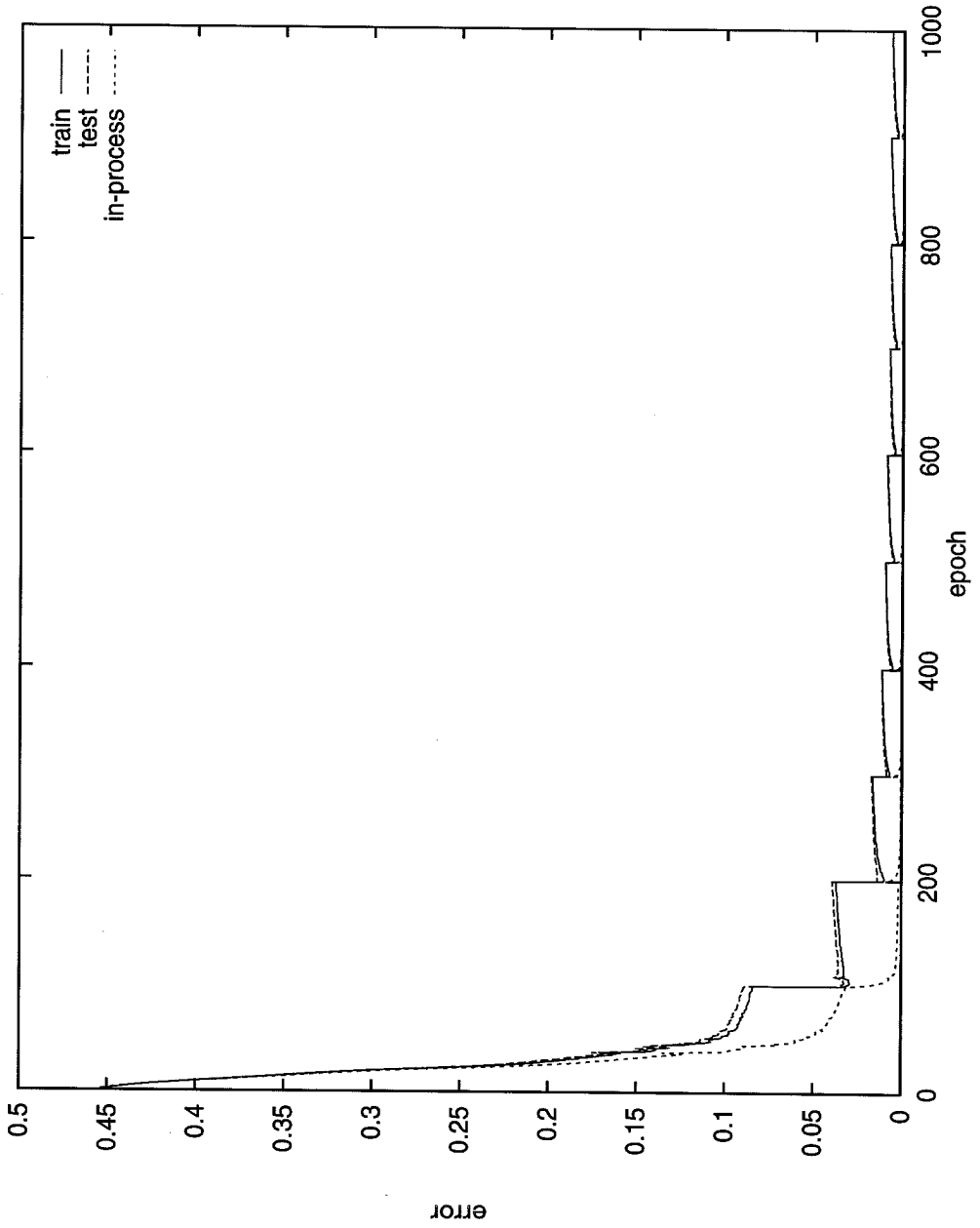

Figure 11.13: Gradient Descent Proportioning – 8 Processes – Partitioned Data – Training Epochs

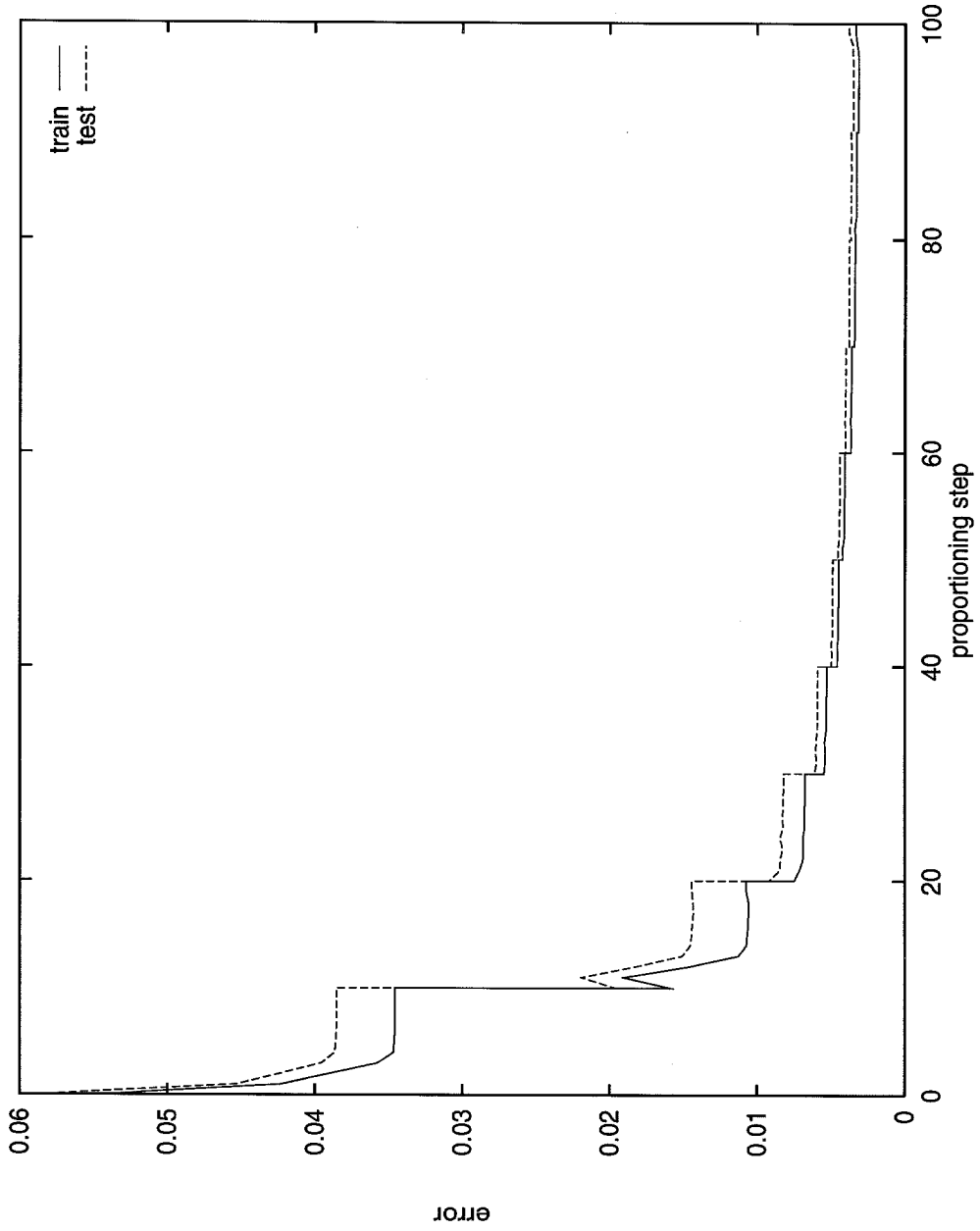

Figure 11.14: Gradient Descent Proportioning – 8 Processes – Partitioned Data – Gradient Descent Steps

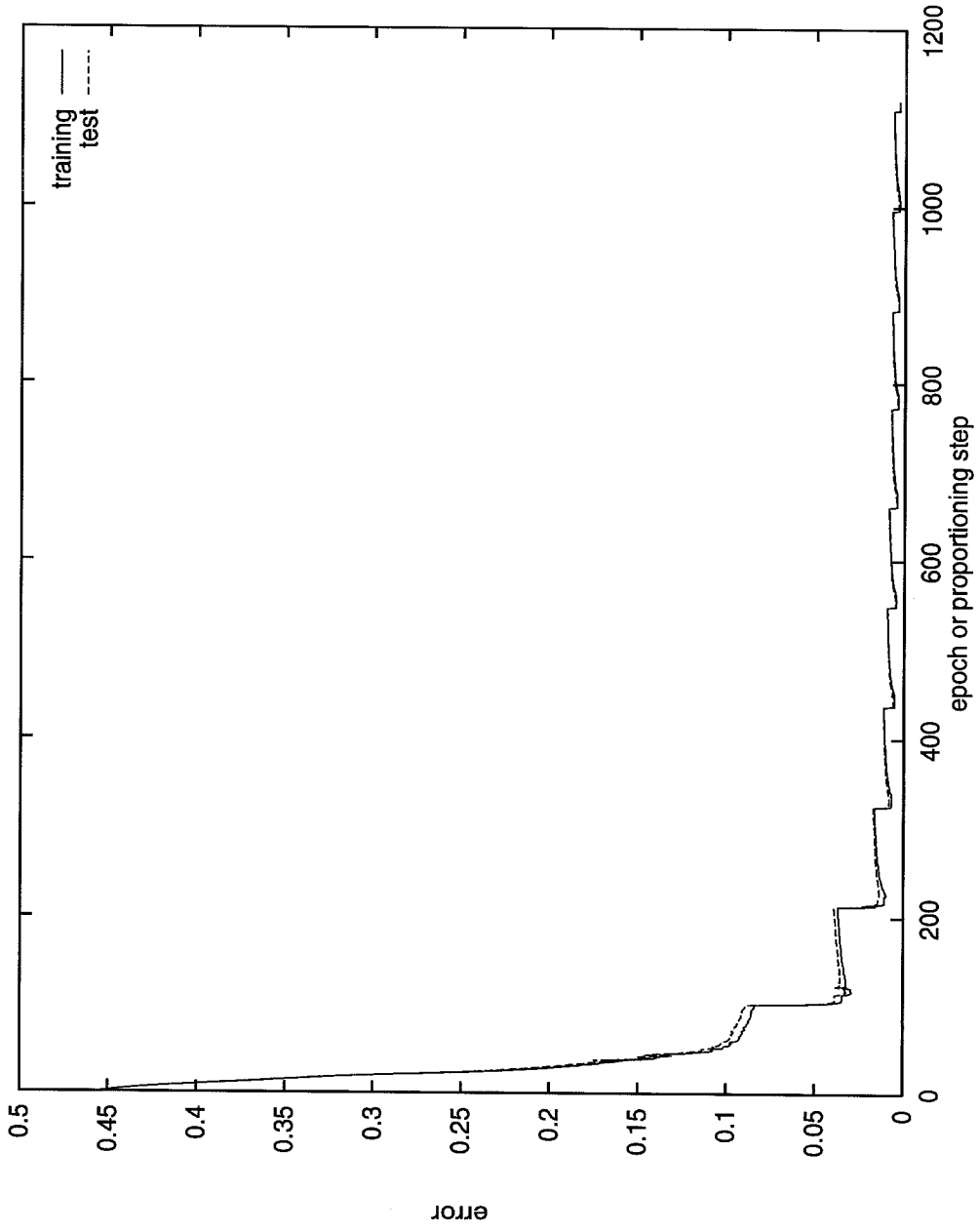

Figure 11.15: Gradient Descent Proportioning – 8 Processes – Partitioned Data – Training Epochs and Gradient Descent Steps

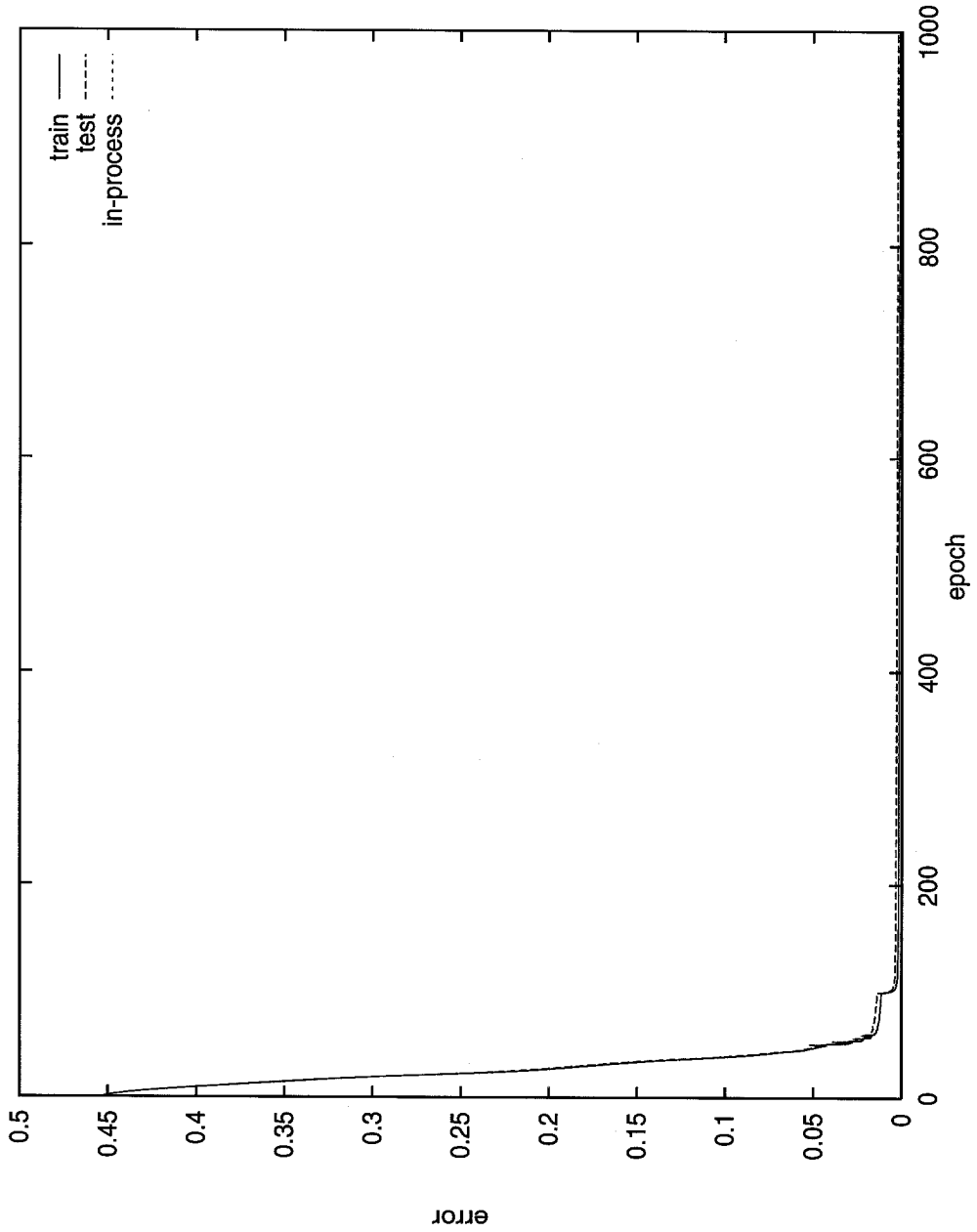

Figure 11.16: Gradient Descent Proportioning – 8 Processes – Duplicated Data – Training Epochs

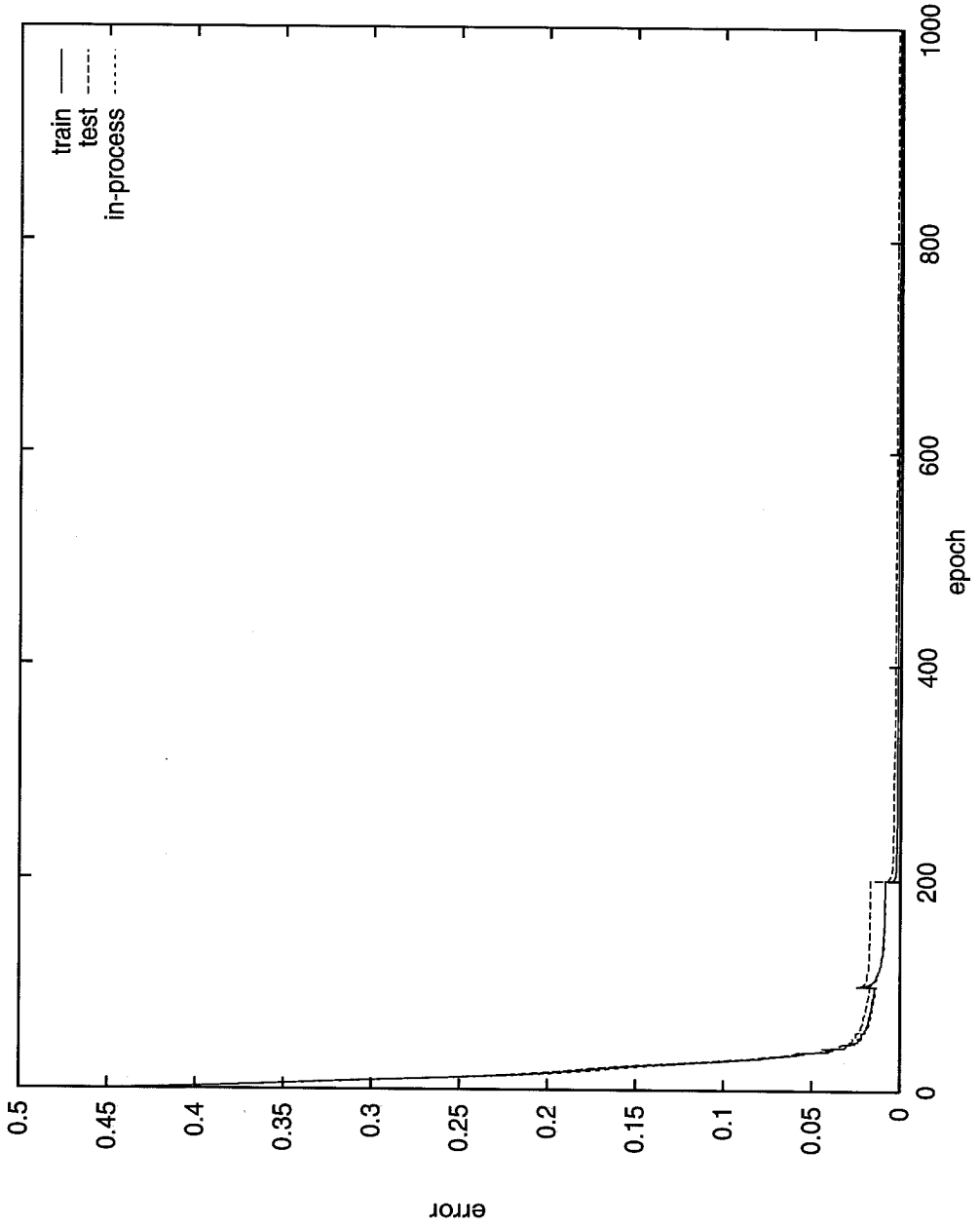

Figure 11.17: Gradient Descent Proportioning – 8 Processes – Orbit Training – Training Epochs

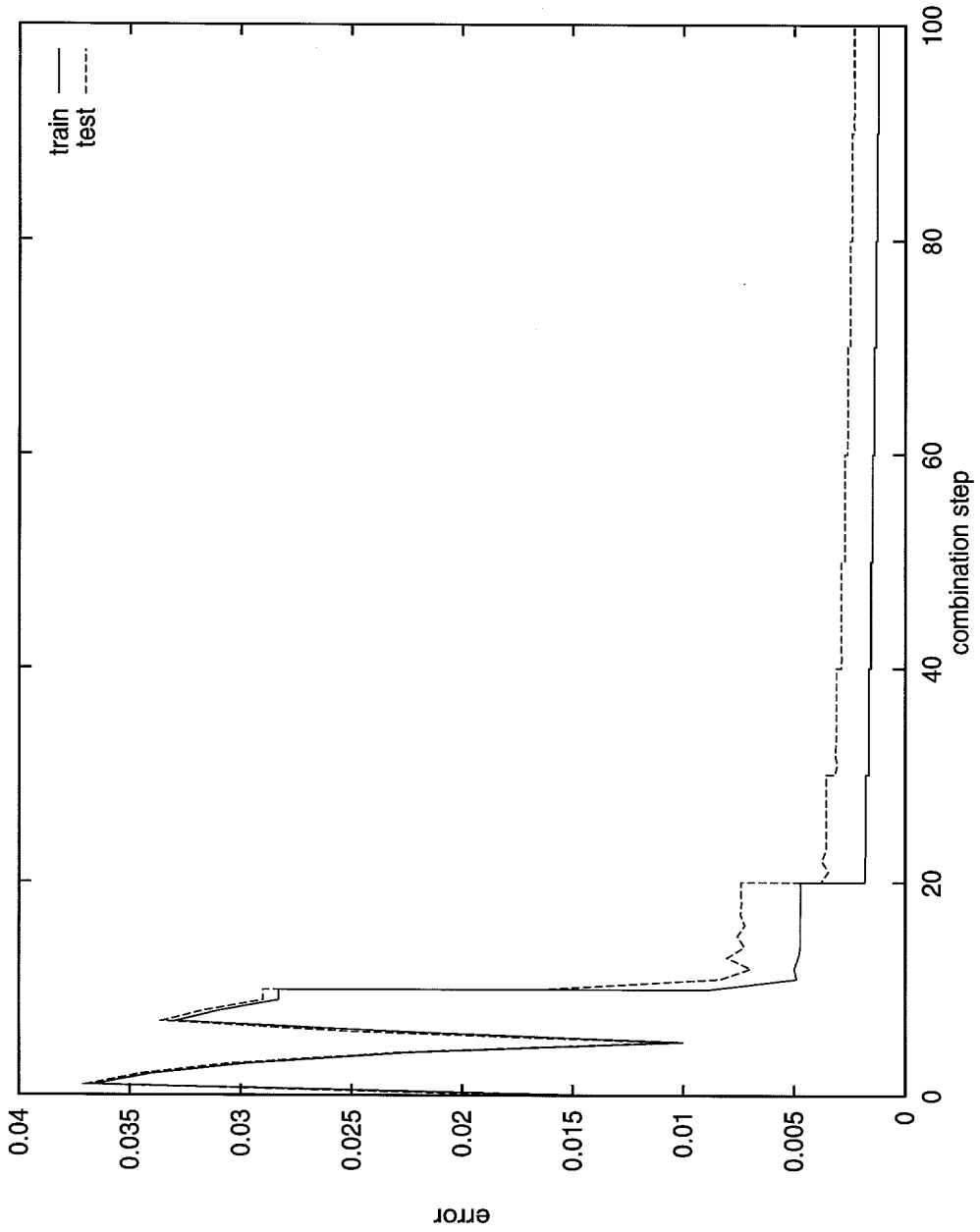

Figure 11.18: Gradient Descent Proportioning – 8 Processes – Orbit Training – Gradient Descent Steps

training epochs faster as more processes are used.

Figure 11.21 shows the evolution of the test error over time spent in computation for the methods that can execute training epochs more quickly as more processes are used – the standard gradient sum method and the combination training methods with training data partitioned among the processes. The curves in Figure 11.21 are derived by stretching curves in Figure 11.19 according to the amount of time their programs spent executing the learning algorithms. (The timings do not measure time spent collecting output or writing output to files; the execution times are measured using versions of the programs with all output collection and writing eliminated.) Figure 11.22 is a close-up of Figure 11.21.

Figure 11.21 shows that the combination method with gradient descent proportioning performed best, while the fan-in combination method performed worst. The combination methods took about the same amount of time to complete 1000 epochs of training. The standard gradient sum method performed each epoch slightly faster.

The interval size (100 epochs of training between combinations) was chosen to make the combination methods take about as much time per epoch as the gradient sum method. A larger interval size means fewer interruptions of training, so the time per epoch decreases. Since training increases test error after the first few intervals, and combination reduces test error, it might seem that combining the networks after fewer training epochs would cause a greater reduction in test error per training interval. However, this is not necessarily true. Training and combination operate in concert to decrease the test error, even when training increases test error. Observe Figure 11.23, which compares the learning speed with combinations every 20 epochs to the learning speed with combinations every 100 epochs. Both curves represent 1000 epochs of training. Note the behavior of the test error after the first 50 seconds. Training for 100 epochs causes the test error to rise more than training for 20 epochs, but then combination after 100 epochs of training causes the test error to decrease more than combination after 20 epochs of training. The net effect is that the larger interval size increases learning per interval. The methods have about the same rate of learning per time (see Figure 11.23.)

Training is necessary to realize the benefits of combination. Without training between combinations, the processes would combine identical networks. So combination could return only a scalar multiple of the network. In the other limit, if there are many processes, and each process has only a single data point, then training each network on its single data point for many epochs would produce networks with so much variety that their hidden units could not be effectively matched by function. There is a tradeoff. With too little variety, combination produces a network that is nearly identical to the networks being combined. With too much variety, the networks cannot be matched well enough for effective combination.

The tests in which the data is partitioned among the processes show that the

combined network can generalize well even when the networks being combined are not trained on enough data to generalize well. Using combination to increase generalization is the subject of the next chapter.

11.5 How to Proceed in Practice

The tests in previous sections expose the behaviors of various methods to train networks on multicomputers. This section outlines how to apply the methods when faced with a particular problem. A problem instance consists of a data set, a network architecture, and a multicomputer. A solution consists of a trained network that performs well on out-of-sample data.

Given a problem, the following points should be addressed:

- *Use interval training and combination.* This training paradigm offers the option of sequential mode training, while the gradient sum procedure requires batch mode training. Also, training with network combination can take advantage of extra process memory to speed training, while the gradient sum procedure cannot be improved by using more memory than is necessary to partition the data among processes.

- *Use fan-out matching and gradient descent proportioning.* The tests in earlier sections show that this method produces faster learning than fan-in combination. Fan-out matching minimizes the maximum lengths of matching chains required to match a set of networks by pairwise matchings. Using gradient descent to proportion trained networks is a natural extension of using gradient descent to train the networks. If the networks are well-trained, then proportioning the networks is a gradient descent search constrained to a subspace of weight space spanned by networks with low error.

- *Place as much data as possible in each process.* If there is enough memory, duplicate the complete data set in each process. If not, then duplicate the complete data set in sets of processes. For example, if each process has enough memory for half of the data, then pair off processes and duplicate the entire data set in each pair of processes. It is better to train for a single epoch on a hundred data examples than to train for five epochs on twenty data examples.

- *Randomize.* Randomize the order of presentation of training examples in each epoch. Set the initial weights of different student networks independently. If the data set is partitioned among sets of processes, use different data splits in the sets.

- *Train in-process for speed. Orbit train for accuracy.* The best situation is to have the data duplicated in every process. Failing that, there is a

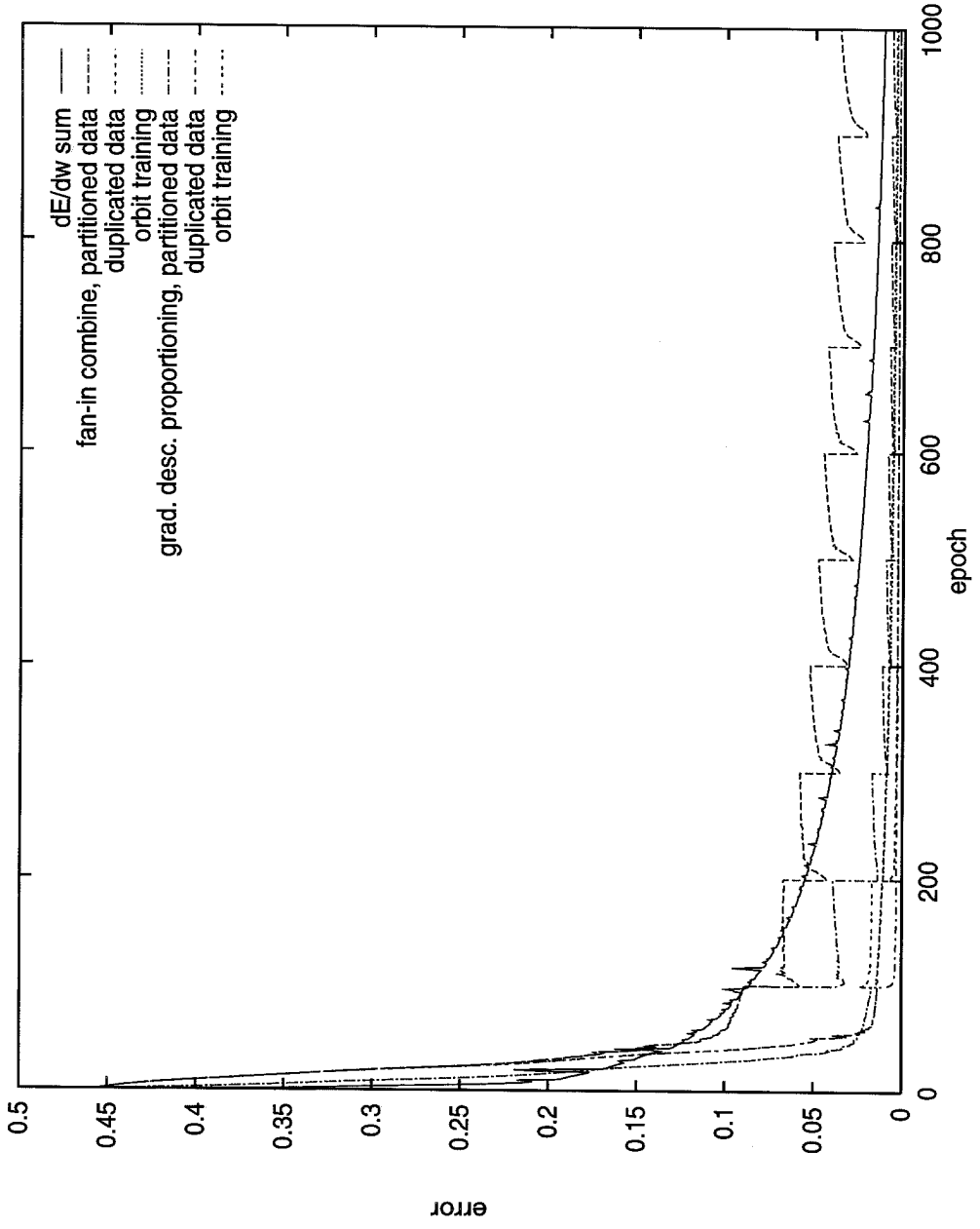

Figure 11.19: Test Errors Over Training Epochs For Various Training Methods

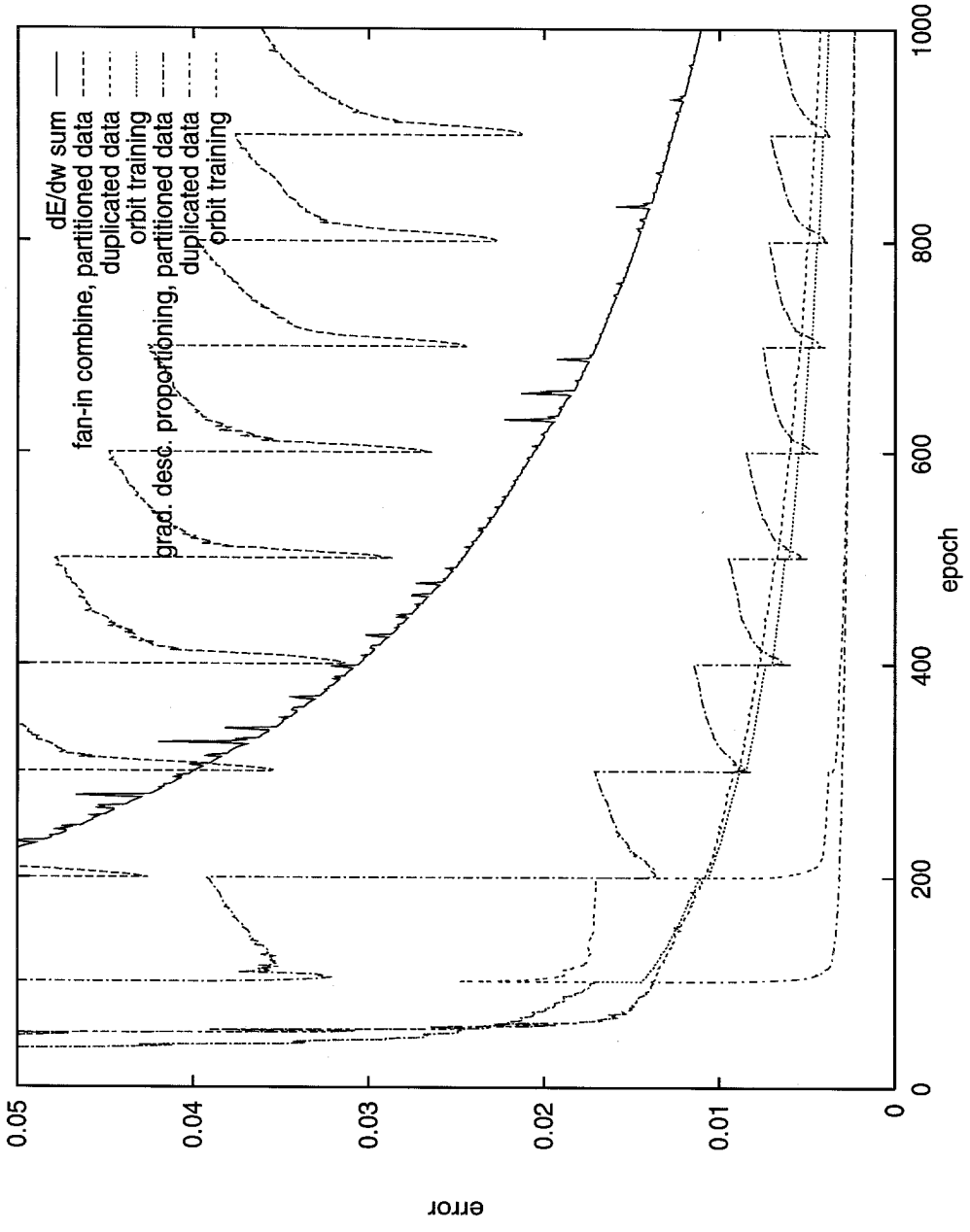

Figure 11.20: Detail of Test Errors Over Training Epochs For Various Training Methods

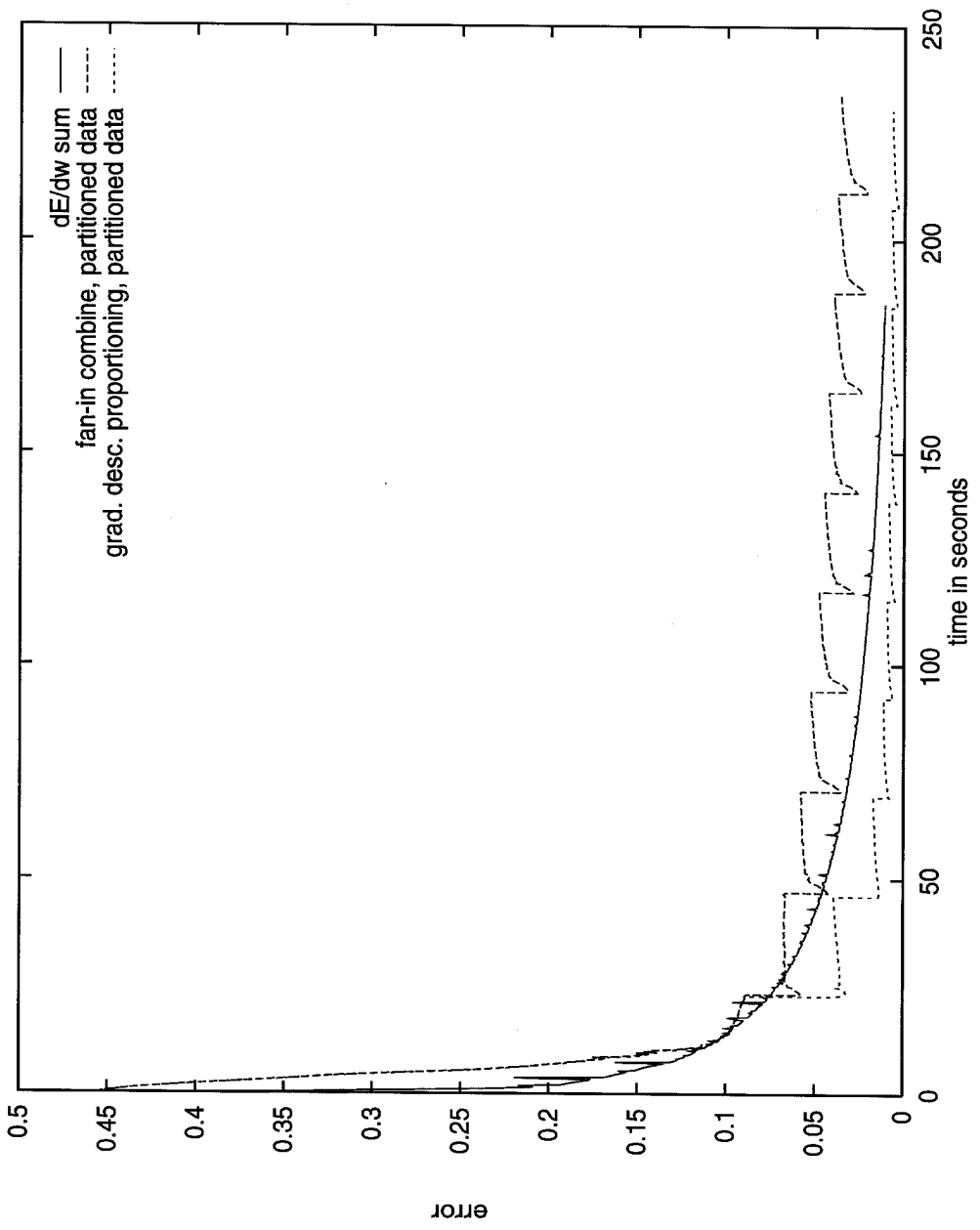

Figure 11.21: Learning Speeds of Various Training Methods on 8 Processes

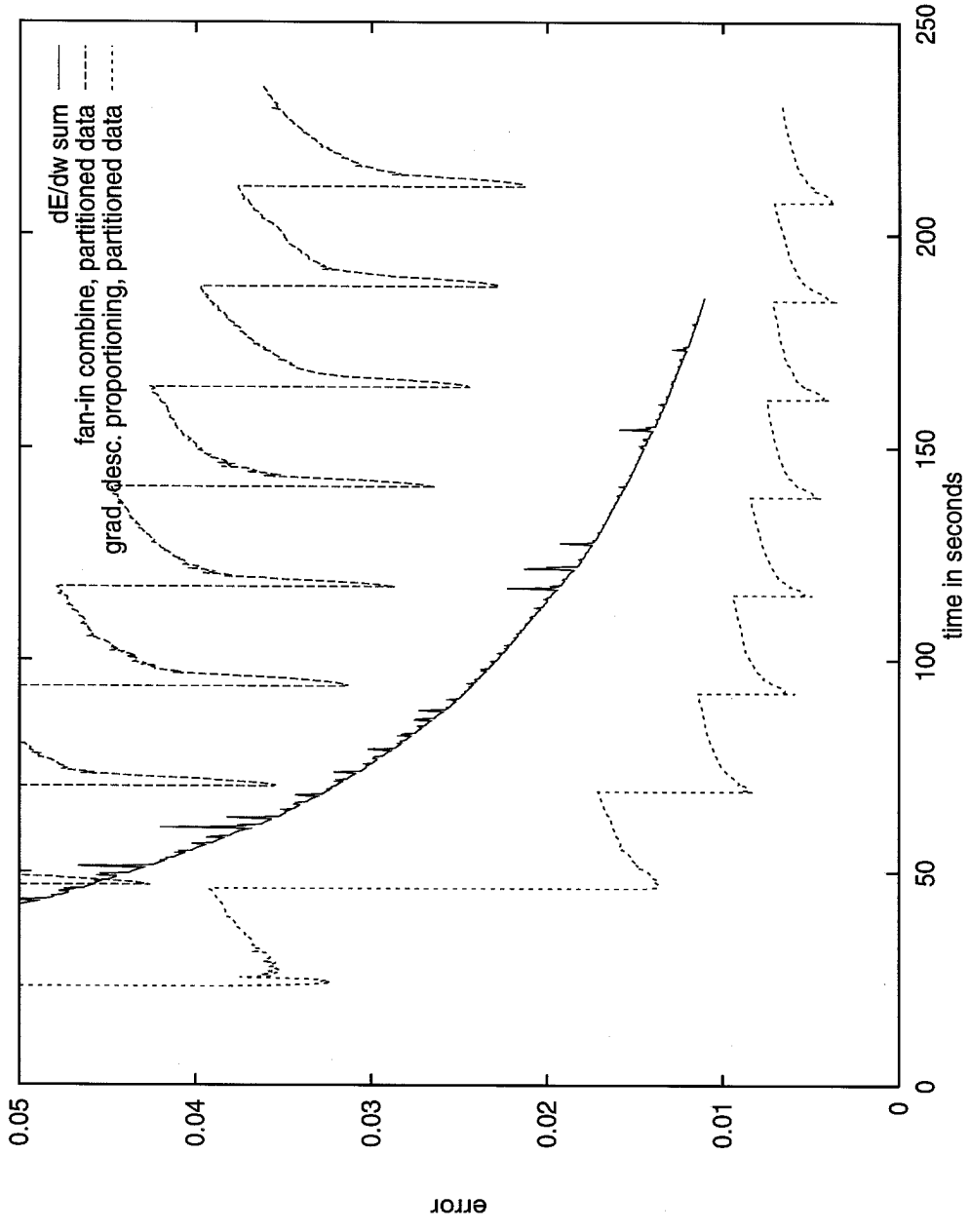

Figure 11.22: Detail of Learning Speeds of Various Training Methods on 8 Processes

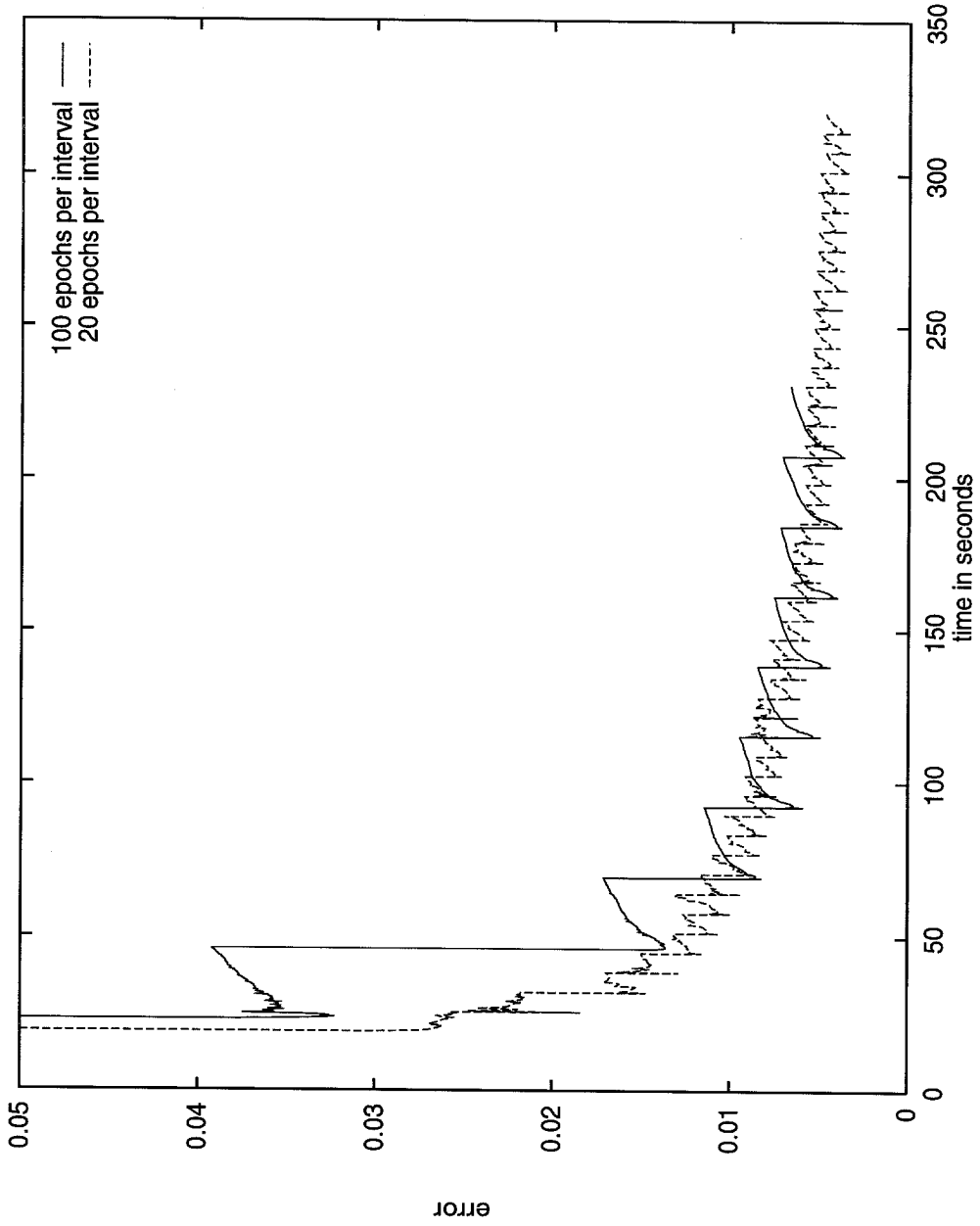

Figure 11.23: Learning Speeds for Different Interval Sizes

tradeoff between orbit training and in-process training. The best course of action may be to use in-process training initially, when any exposure to data reduces network error. Then switch to orbit training for the later epochs, to polish off the learning.

11.6 Conclusion

We have shown that weight-by-weight convex combinations of a pair of networks generally have lower test error than either network in the pair if one of the networks is rearranged by HUF or BNF matching so that corresponding hidden units play corresponding roles in the networks' functions, and the networks are trained on data sets with examples of the same function, drawn from the same input distribution. HUF and BNF evaluate the similarity of hidden units by observing the roles played by the hidden units while the networks operate on data examples that reflect the input distribution of out-of-sample data. Hence, these matching methods emphasize similarity over the regions of input space that will be encountered most in practice. This makes HUF and BNF inherently more robust than matching methods that evaluate similarity by comparing network weights alone.

We have shown that combination without matching fails. We have shown that converting both networks to a canonical form and then matching also fails. The transformation to canonical form is sufficient to match networks that perform the same function, but the method is not robust enough to match networks that perform similar functions.

We have shown that HUF or BNF matching and combination are successful even after training a pair of networks separately for many epochs. We have also shown that the following training strategy is viable – train a pair of networks on separate data sets for several epochs, match and combine the networks, replace each network in the pair by the combination network, and repeat. This strategy is the basis of a multicomputer training method that has less communication overhead than the current batch mode strategy.

11.7 Acknowledgements

I thank Yaser Abu-Mostafa for his advice and guidance concerning the development of these results. I thank Zehra Cataltepe, Malik Magdon-Ismail, Joseph Sill, and Xubo Song for many helpful pointers and educational conversations.

Bibliography

[1] Y. S. Abu-Mostafa, personal communication.

[2] Zehra Cataltepe and Malik Magdon-Ismail, personal communication.

[3] K. M. Chandy and J. Misra, *Parallel Program Design, A Foundation.* (Addison-Wesley, 1988)

[4] A. M. Chen, H. Lu and R. Hecht-Nielsen, On the geometry of feedforward neural network error surfaces, *Neural Computation* 5 (1993) 910-927.

[5] H. W. Kuhn, The Hungarian method for the assignment problem, *Naval Res. Logist. Quart.* 2 (1955) 83-97.

[6] B. M. E. Moret and H. D. Shapiro, *Algorithms from P to NP, Volume I.* (The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1990)

[7] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization, Algorithms and Complexity* (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982) 433-448.

[8] H. Paugam-Moisy, Optimal speedup conditions for a parallel back-propagation algorithm, *Lecture Notes in Computer Science* 634 (1992) 719-724.

[9] D. A. Pomerleau, G. L. Gusciora, D. S. Touretzky and H. T. Kung, Neural network simulation at Warp speed: How we got 17 million connections per second, *IEEE Internat. Conf. on Neural Networks,* San Diego, CA, 1988, pp. 143-150.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, Second Edition* (Cambridge University Press, New York, NY, 1992) 379-393.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning internal representations by error propogation, Ch.8 in *Parallel Distributed Processing, Volume 1: Foundations.* (The MIT Press, Cambridge, MA, 1986)

[12] N. Shamir, D. Saad and E. Marom, Neural net pruning based on functional behavior of neurons, *International Journal of Neural Systems* 4 (1993) 143-158.

[13] N. Shamir, D. Saad and E. Marom, Preserving the diversity of a genetically evolving population of nets using the functional behavior of neurons, *Complex Systems* 7 (1993) 327-346

[14] N. Shamir, D. Saad and E. Marom, Using the functional behavior of neurons for genetic recombination in neural nets training, *Complex Systems* 7 (1993) 445-467

[15] J. Sill, Monotonicity hints, *to appear in NIPS 9*

[16] A. Singer, Implementations of artificial neural networks on the Connection Machine, *Parallel Computing* 14 (1990) 305-315.

[17] H. J. Sussman, Uniqueness of the weights for minimal feedforward nets with a given input-output map, *Neural Networks* 5 (1992) 589-593.

[18] E. F. Van de Velde, *Concurrent Scientific Computing* (***Publisher?***)

[19] M. Witbrock and M. Zagha, An implementation of backpropagation learning on GF11, a large SIMD parallel computer, *Parallel Computing* 14 (1990) 329-346.

[20] X. Zhang, M. McKenna, J. P. Mesirov, and D. L. Waltz, The backpropagation algorithm on grid and hypercube architectures, *Parallel Computing* 14 (1990) 317-327.